Graph transformation and designing parallel sparse matrix algorithms beyond data dependence analysis

H.X. Lin
Delft Institute of Applied Mathematics (*DIAM*), *Delft University of Technology, 2628 CD, Delft, The Netherlands*
E-mail: H.X.Lin@ewi.tudelft.nl

Abstract. Algorithms are often parallelized based on data dependence analysis manually or by means of parallel compilers. Some vector/matrix computations such as the matrix-vector products with simple data dependence structures (data parallelism) can be easily parallelized. For problems with more complicated data dependence structures, parallelization is less straightforward. The data dependence graph is a powerful means for designing and analyzing parallel algorithms. However, for sparse matrix computations, parallelization based on solely exploiting the existing parallelism in an algorithm does not always give satisfactory results. For example, the conventional Gaussian elimination algorithm for the solution of a tri-diagonal system is inherently sequential, so algorithms specially for parallel computation has to be designed. After briefly reviewing different parallelization approaches, a powerful graph formalism for designing parallel algorithms is introduced. This formalism will be discussed using a tri-diagonal system as an example. Its application to general matrix computations is also discussed. Its power in designing parallel algorithms beyond the ability of data dependence analysis is shown by means of a new algorithm called ACER (Alternating Cyclic Elimination and Reduction algorithm).

Keywords: Parallel matrix algorithm, graph transformation, unifying graph model

1. Introduction

Efficient parallelization of a computational problem requires resolving the tradeoffs between maximal load balance and minimal communication overhead. The fundamental factor limiting load balance is the parallelism in an algorithm. Communication costs can often be reduced by increasing data locality through appropriate partitioning of data and computations. In this paper we will focus on how to introduce parallelism. Because it is known that the conventional Gaussian elimination process for the tri-diagonal system has very little parallelism, so the problem of solving a tri-diagonal system of linear equations is a good example for demonstrating the power of the graph formalism.

Several parallelization approaches can be distinguished which deal with from simple to more complicated data dependence structures. 1. In the data parallel approach, the result data are partitioned into groups and each processor is responsible for the computation associated with a single group of the result. These groups can be computed in parallel (independent from the results of each other). 2. For problems with more complicated data dependence relations, some analysis has to be performed to detect the parallelism in the algorithm. Note that without giving a formal definition we roughly distinguish an algorithm and a program by that an algorithm has only data dependence (i.e. (input) data availability) as concerns and whereas a program may have additional dependency (e.g. due to the use of intermediate variables). We will focus on algorithms with ‘pure’ data dependences and are not concerned with output dependence, anti-dependence and loop exchange etc. (see Kuck et al. [6]). In general, parallel compilers analyze the different type of dependences, and try to reduce or eliminate the output dependence or anti-dependence by performing some transformation on a given program (e.g. [14,17]). However, the 'pure' data dependence in the program must be obeyed and therefore transformation techniques of the parallel compilers cannot increase parallelism constrained by the data dependence in a given program. 3. The aforementioned two approaches are based on exploiting the existing parallelism (i.e. allowed by data dependence) in a given algorithm. Sometimes it is advantageous or even necessary to modify the algorithm in order to obtain an efficient parallelization. An example is the factorization of sparse matrix. Often an ordering for the elimination sequence is first determined on the so-called elimination graph. An elimination of a node in this undirected graph corresponds to the elimination of an entire column below the main-diagonal of the matrix. The elimination graph is widely applied in ordering for both minimizing the number of fill-ins and for parallelism in the matrix factorization process.

Graph models have been used for parallel algorithms and for sparse matrix computations. The acyclic directed graph (DAG) is often used to represent the data dependences between the operations in a (parallel) algorithm. A (directed) edge (a, b) represents the input-output data dependence among operation a and b. Both potentials and restrictions on parallelism can be analyzed with the DAG. The difference between the DAG and the elimination graph lies in that the DAG represents an algorithm whereas the elimination graph represents the factorization problem. Different orderings (of the elimination graph) correspond to different algorithms.

Recently, we have introduced a more general graph formalism for sparse matrix computations [10,11] which uses a directed graph and the basic operation is the elimination of one edge (in contrast to the elimination of a node with all the edges incident on it in the conventional elimination graph). It is based on the concept of viewing a sparse matrix computation problem as a graph transformation problem: the initial graph representing the initial matrix is transformed into a terminal configuration representing the desired final matrix form (e.g., a triangular matrix, or a diagonal matrix, etc.). The transformations are governed by a set of rules which prescribe the transformed graph is equivalent with the initial graph. Similarly additional rules can be defined to determine which operations can be done in parallel.

It is shown that the common elimination operations and factorization operations are special cases of graph transformations in our formalism. The power of the formalism is demonstrated by applying it to a number of known parallel algorithms for tridiagonal systems, such as the cyclic reduction, the recursive doubling, the block partitioned elimination algorithms, and the block partitioned (Cholesky) factorization algorithms. The presented formalism not only provides us a systematic way to describe existing parallel algorithms, but also a means to design and optimize them.

In Section 2, we will first describe the relationship between elimination operations and graph transformations, and introduce the graph framework for analyzing and designing parallel matrix algorithms. In Section 3, we show that a number of known parallel algorithms can be unified by the framework. We also show how to use the framework for generating parallel algorithms with a certain desired property. Section 4 describes a new class of parallel algorithms for tri-diagonal matrices, and Section 5 concludes the paper.

2. Gaussian elimination and graphs

First, we review the relationship between graph transformation and Gaussian elimination for a general matrix. Then, we proceed to discuss the basic relations between parallel elimination of edges and fill-ins. A number of properties of graph transformation and parallel elimination will be investigated. These form the basis for defining the selection, elimination and update operations which are used to transform an initial graph into a graph with the required property.

The finite-difference and finite-element grids are graphs which often naturally correspond to the coefficient matrix of the linear system of equations. Many researches have been done to minimize the number of fill-ins.

In [15] Parter has introduced the use of graphs for determining fill-ins[1] in a sparse matrix during the Gaussian elimination. He studied the relationship between a non-zero in the matrix and the addition of an edge in the adjacency graph. An adjacency graph or elimination graph associated with matrix A is an undirected graph with the set of edges defined as $E = \{(i,j) \mid a(i,j) \neq 0 \text{ or } a(j,i) \neq 0\}$. Note that we use the term *edge* and *arc* for *undirected* respectively *directed* connections between two nodes.

It is shown that the elimination of a variable i only changes the coefficients $a(k,j)$ in the matrix if and only if there exist a pair of edges (k,i) and (i,j) [15].

[1]A fill-in is a coefficient which is zero in the original A, but become nonzero during the elimination/factorization.

That means for a sparse matrix the modification in coefficients is local. Since a fill-in during the elimination or factorization corresponds to the addition of a new edge in the elimination graph, so minimizing fill-ins is the same as minimizing the number of additional new edges in the elimination graph. This property is often used in determining an ordering with minimum number of fill-ins for a sparse symmetric matrix [2,3,18].

With respect to parallel elimination, an important fact is that two non-adjacent nodes in an elimination graph can be eliminated independently. A proof of this for symmetric matrices (corresponding to an undirected graph) can be found in [8]. Peters [16] has studied the parallel pivoting algorithms for sparse symmetric matrices. He noted that given an ordering (i.e., elimination sequence), two pivots i and j can be eliminated in parallel if there does not exist any path between them which comprises of solely nodes ordered before i and j. He described implementation procedures to exploit the parallelism in the pivoting process. A self-ordering procedure is also presented in [16] which does not assume an a priori ordering of the matrix or graph. At an elimination step, two nodes in the (remaining) elimination graph can be eliminated in parallel if they are not adjacent. In [8] Lin studied the use of quotient elimination graphs for parallel factorization of block structured sparse matrices obtained from domain decomposition of finite element meshes. His work is concerned with coarse-grain block parallelism in contrast to the single node pivoting. These aforementioned researches deal with sparse symmetric matrices and undirected graphs. It is sufficient for the problem of applying standard Gaussian elimination or to compute the Cholesky factorization, in which we eliminate the matrix coefficients below the diagonal one column at a time. This corresponds to eliminating a node and all the edges connected to it from the elimination graph.

The undirected elimination graph has been successfully used for minimizing fill-ins in sequential solution (e.g. [2,3]) and parallel factorization of sparse symmetric matrices. However, it cannot describe many operations in a parallel matrix algorithm. For instance the recursive doubling algorithm [19] and the partition method [20] cannot be described in terms of eliminating each column or row exactly once during the elimination process. In the partition method some rows or columns are modified several times. Unlike in Gaussian elimination or LU-factorization the final form is here not an upper triangular matrix. Therefore, we use directed graphs in our framework. In order to obtain the required high flexibility we study the parallelism in eliminating an arc (j,i) associated with $a(j,i)$ using $a(i,i)$ instead of eliminating an entire column i. Note that we don't assume $i > j$ or $j > i$ here, i.e., we don't assume any pre-determined elimination ordering. We consider the elimination of an arc as an operation which can be performed in any order or even repeatedly. Throughout the paper, we ignore numerical cancellations when considering fill-ins and fill-arcs during an elimination process. An non-zero is thus a logical non-zero and an accidental numerical cancellation is not considered as a zero coefficient.

Given an $n \times n$ matrix A, a corresponding directed graph $G(V,E)$ is defined as a graph with the set of nodes $V = \{1,2,\ldots,n\}$ and the set of arcs $E = \{(i,j) \mid a(i,j) \neq 0, i \in V \wedge j \in V\}$. Arc (i,j) is said to have a *begin node* i and *end* (or *terminal*) *node* j. (i,j) is an outgoing arc from i, and an incoming arc to j. The set of predecessors and successors of node i are denoted by $PRED(i)$ and $SUCC(i)$. As a convention, in this paper we define that the node i is never a predecessor or successor of itself.

Theorem 1. *Consider the elimination of arc (i,j) using node j, arc (i,k) exists if and only if (i,k) or (j,k) exists before the elimination.* $\square$

Proof. Elimination of arc (i,j) corresponds to using row j to eliminate the entry $a(i,j)$ in the matrix. This means adding $-\frac{a(i,j)}{a(j,j)} \cdot (row\ j)$ to $(row\ i)$, which yields $a^{new}(i,k) = a(i,k) - \frac{a(i,j)}{a(j,j)} \cdot a(j,k)$. Because $\frac{a(i,j)}{a(j,j)} \neq 0$, so $a^{new}(i,k) \neq 0$ if and only if $a(i,k) \neq 0$ or $a(j,k) \neq 0$. QED.

The graph corresponding to a tridiagonal matrix is a bidirectional linear chain (Fig. 1 a). The standard Gaussian elimination algorithm eliminates edge $(k+1,k)$, $k = 1,2,\ldots,n-1$, at step k during the (forwards) elimination process. After the (forwards) elimination, a graph with a path starting from node n, through $n-1$, $\ldots$, to node 1 results (Fig. 1 b). In this graph there are only edges (i,j) with $i = j$ or $j = i+1$, this corresponds to an upper triangular bidiagonal matrix. The unknowns can now thus be computed through back-substitution, starting from n, through $n-1$, $\ldots$, to 1. This results in a graph of n disconnected nodes (which corresponds to a diagonal matrix). Notice that the standard Gaussian elimination process is inherently sequential, only one arc at a time is eliminated.

The corresponding numerical operations on the matrix can be defined on this graphical representation. We denote the coefficient $a(i,j)$ as the weight of arc (i,j),

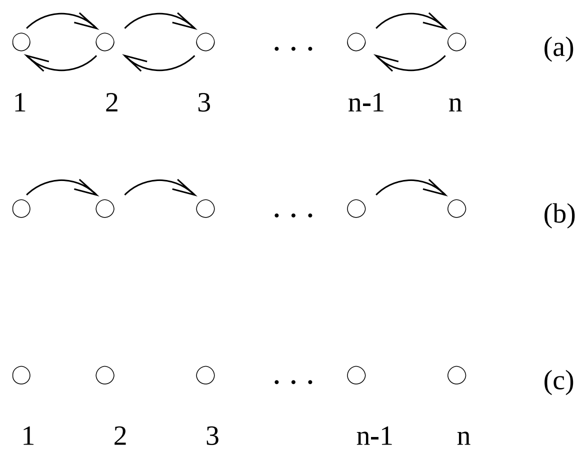

Fig. 1. Graph transformations corresponding to the Gaussian elimination.

an non-existing arc is equivalent to an arc having zero weight. The elimination of arc (i, j) will correspond to the following operations in the graph:

- for each $k \in SUCC(j)$: $a(i, k) = a(i, k) - a(i, j) * a(j, k)/a(j, j)$. (Note that if $a(i, k)$ was 0, i.e., arc (i, k) does not exist, then (i, k) is a fill-arc).
- remove arc (i, j);

Since we are only interested in the parallelism and the structure of the parallel operations in this paper, we will omit the discussion of the numerical operation of the coefficients. The weight of the arcs is also omitted except that we define there is an arc when $a(i, j)$ is logically non-zero, and (i, j) does not exist when $a(i, j) = 0$.

Lemma 1.1. *The elimination of arc (i, j) results in arcs (i, k) for all successors of j, i.e., after the elimination we have (i, k) for each $k \in SUCC(j)$. These are the only modifications in the edge-connectivity as a result of the elimination of (i, j).* $\square$

Lemma 1.1 follows straightforward from Theorem 1. It states that a fill-arc (i, k) occurs when eliminating arc (i, j) if and only if there is a path $i \to j \to k$, and (i, k) does not exist before the elimination of (i, j). Consequently the elimination of all incoming arcs to node i will result in all predecessors of i connected to all successors of i. The elimination of all outgoing arcs (i, j) from node i, result in node i become connected to all nodes in $SUCC(j)$ for each $j \in SUCC(i)$. In matrix terminology, the elimination of all incoming and outgoing arcs of node i corresponds to the elimination

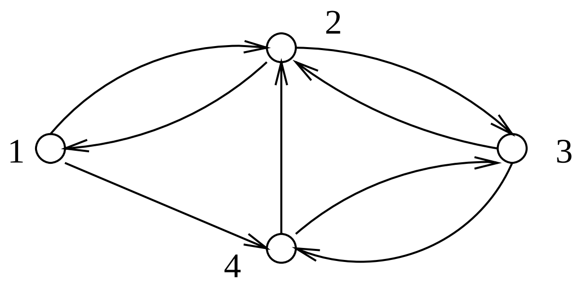

Fig. 2. An example graph.

of all off-diagonal non-zeros of column i and row i respectively. Lemma 1.1 also tells us the so-called 'path-conservation' principle of the transformation: if k_1 and k_2 are connected then k_1 and k_2 remains connected when an arbitrary arc (i, j) in the graph is eliminated provided that $(k_1, k_2) \neq (i, j)$.

Lemma 1.2. *Parallel elimination of (i_1, j_1) and (i_2, j_2) is successful if and only if*

1. *$i_1 \neq i_2$; or*
2. *$i_1 = i_2$ and j_1 is not in $SUCC(j_2)$ and j_2 not in $SUCC(j_1)$.* $\square$

Lemma 1.2 can be proved directly using Theorem 1. We define a *parallel elimination step* as the elimination of a set of arcs which can be eliminated independently. A parallel elimination of two arcs is said to be successful if they are both eliminated after the parallel elimination step. This is of course not always possible because the elimination of the arc (i_2, j_2) may cause the return of (i_1, j_1) (although now with another value of $a(i_1, j_1)$), and vice versa. Lemma 1.2 tells us that parallel eliminations of two arcs initiating from the same begin node can only be successful if there is no arc between the terminal nodes of these two arcs. Take for example Fig. 2, the set of arcs {(1,2), (2,1), (2,3), (3,2), (4,3)} can be successfully eliminated in parallel, but the pair (1,2) and (1,4) cannot be eliminated in parallel.[2] This conclusion can easily be extended to a set of arcs.

Consider a directed graph $G(V, E)$ associated with an $n \times n$ matrix. An algorithm which determines the parallel elimination of arcs in G until all nodes become isolated is given in Algorithm 1. The elimination of the arcs in each set $S(k)$ comprises one elimination

[2]they can be eliminated in the order of first (1,4) and then (1,2), but not first (1,2) followed by (1,4), in the latter case the arc (1,2) will return.

```
Initialize $G(V,E)$ with $V = \{1,2,...,n\}$ and
        $E = \{(i,i+1)|i = 1,...,n-1\} \cup$
            $\{(i,i-1)|i = 2,...,n\}$;
$k = 1$;
WHILE $E \neq \emptyset$ DO
    SELECT all possible arcs $S(k)$ for parallel
        elimination (Lemma 1.2);
    ELIMINATE $S(k)$ from $E$;
    UPDATE $E$ according to Lemma 1.1;
    $k = k + 1$;
END
```

Fig. 3. Algorithm 1 – A greedy algorithm for parallel elimination.

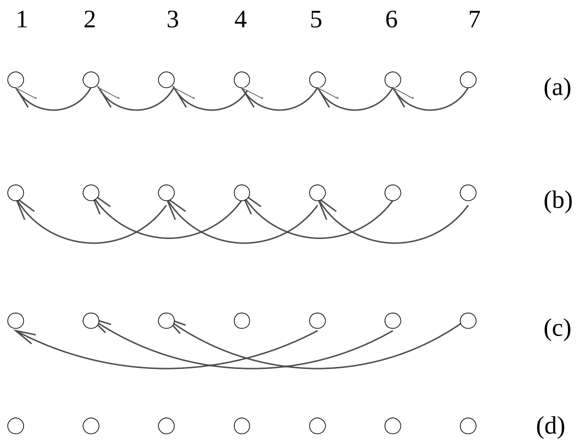

Fig. 4. Application of Algorithm 1 to a bi-diagonal system leads to the recursive doubling scheme.

step. Note that other possible end conditions are: each node in the graph has only either outgoing or incoming arcs, the remaining graph forms a single or multiple spanning tree(s). This is a fundamental difference from the case of undirected graphs where the end configuration is always a triangular matrix (corresponding to all nodes are eliminated from the elimination graph). Another fundamental difference between the presented directed graph model and the conventional undirected graph is that in case of a directed graph model a parallel elimination step is generally not divisible into two or more equivalent sub-steps (as will be discussed later in the section about update conflicts).

The key is the Selection operation. Algorithm 1 is a greedy algorithm in which at each parallel elimination step as many as possible arcs are selected for parallel elimination. Figure 4 shows that the application of the greedy algorithm to a bi-diagonal system of equations (linear recurrence) resulting in a parallel algorithm which is the known recursive doubling algorithm [19]

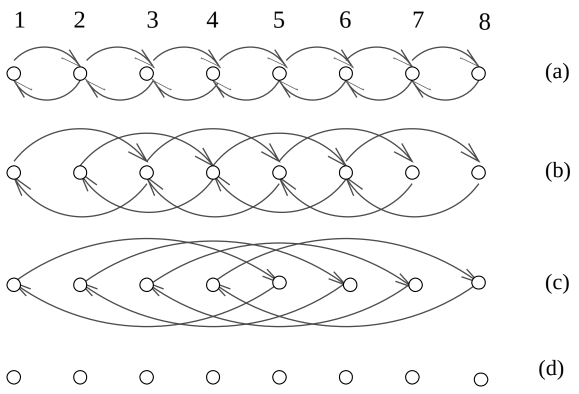

Fig. 5. Illustration of applying Algorithm 1 to a tridiagonal system.

for first order linear recurrence. At the first elimination step, all arcs $(i+1,i)$ for $i = 1,2,\dots,(n-1)$ can be eliminated in parallel. According to Lemma 1.1 the elimination of $(i+1,i)$ causes a fill-arc $(i+1,i-1)$. In the second elimination step, all arcs $(i+1,i-1)$ are eliminated in parallel and fill-arcs $(i+1,i-3)$ are added. In general, at elimination step k, all arcs $(i+1,i-2^{k-1}+1)$ are eliminated in parallel and fill-arcs $(i+1,i-2^k+1)$ are added. This is exactly what the recursive doubling algorithm does for a bidiagonal system.

Figure 5 shows the results of applying the algorithm to an example. Compared to the cyclic reduction algorithm, it can be observed that the number of parallel elimination steps achieved with the greedy algorithm is smaller. As we will show later on the time complexity of this greedy algorithm is $6\log_2(n)+1$ versus $11\log_2(n)-11$ for the cyclic reduction algorithm. However, in some of the steps the elimination of several arcs initiating from the same node causes update conflict. A parallel update conflict occurs when the same coefficient is modified/updated more than once in a parallel elimination step. For example, the elimination of the pair of arcs (2,1) and (2,3) in Fig. 2 has conflict in the update of the coefficient $a(2,2)$.

The next theorem states the conditions when the parallel elimination of arcs (i_1,j_1) and (i_2,j_2) is free of update conflict.

Theorem 2. *The elimination of arcs (i_1,j_1) and (i_2,j_2) are free of update conflict if*

1. *$i_1 \neq i_2$; or*
2. *if $i_1 = i_2$ and $SUCC(j_1) \cap SUCC(j_2) = \emptyset$.* $\square$

Proof. According to Lemma 1.1 the only entries which are modified as a result of the elimination of the arcs

(i_1, j_1) and (i_2, j_2) are the entries associated with the arcs in the sets $S_1 = \{(i_1, k) \mid k \in SUCC(j_1)\}$ and $S_2 = \{(i_2, l) \mid l \in SUCC(j_2)\}$ respectively. Therefore an entry (i, j) is updated more than once only if $(i, j) \in S_1$ and $(i, j) \in S_2$. This means, the elimination of the arcs (i_1, j_1) and (i_2, j_2) is free of update conflict if and only if $S_1 \cap S_2 = \emptyset$. 1. If $i_1 \neq i_2$ then $S_1 \cap S_2 = \emptyset$; and 2. When $SUCC(j_1) \cap SUCC(j_2)$ is empty, so is the set $S_1 \cap S_2$. QED.

If we require in each parallel elimination step the elimination of arcs to be free of update conflict, we can add the above update conflict test in the selection operation of Algorithm 1. That is *"SELECT a set of arcs S(k) where for each pair (i_1, j_1) and (i_2, j_2) in the set the conditions of Lemma 1.2 and Theorem 2 are satisfied"*. Applied with this modification to the example in Fig. 5, the result of parallel eliminations free of update conflict is shown in Fig. 6. It can be observed that the number of elimination steps is increased to 5 in Fig. 6 compared to 3 in Fig. 5. However, there are update conflicts in the first two steps in Fig. 5 which implies a larger parallel execution time in these two steps. Furthermore, we observe that by simply split the first step of the algorithm in Fig. 5 into two sub-steps without update conflict, e.g., first eliminate all arcs $(i, i+1)$ and then all arcs $(i, i-1)$, will result into a totally different (in this case unfavorable) algorithm. In general, a splitting of a parallel elimination step into more sub-steps will result in a different parallel algorithm. This is essentially different from the case of parallel factorization of a symmetric matrix modeled using undirected graphs (e.g. [8]), where the split of a parallel factorization step into several (sequential) sub-steps results in the same algorithm (i.e., fill-ins and update operations are unchanged).

The tests of parallel elimination (Lemma 1.2) and the test of update conflict (Theorem 2) can be simplified by allowing each node being an initiating node at most once in a single parallel elimination step. Furthermore, additional conditions in selecting arcs for parallel elimination is sometimes preferred or required. Such additional conditions can limit the number of fill-arcs, or requiring the total number of fill-arcs must be smaller than the number of arcs being eliminated in each step (the latter is to ensure the finiteness of the elimination process). Additional conditions or heuristics in the selection can also be used for more regularity and control on the parallelism. In fact many of the known parallel algorithms in literature corresponds to applying a certain heuristics or imposing some structure in the elimination process. The general algorithm can be described as consisting of three basic type of operations: 1. partition; 2. selection; and 3. elimination and update.

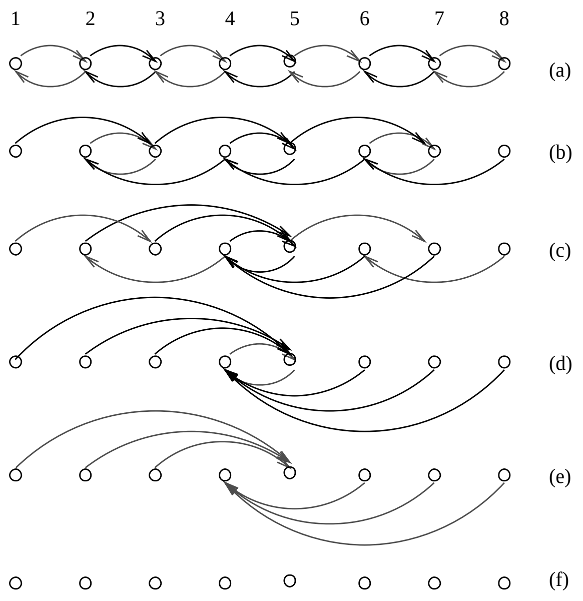

Fig. 6. A parallel elimination scheme free of update conflict.

3. Application to (existing) parallel tri-diagonal algorithms

The solution of tri-diagonal systems occurs in many engineering and scientific computer applications, for instance, it is often a part of the solution process in numerical simulation of PDEs using finite difference or finite element discretization. On the other hand, when standard Gaussian elimination method is applied to a (pre-ordered) tri-diagonal matrix, the computation is inherently sequential. That is why it has inspired many researchers to study parallel algorithms for solving this problem (in the past three decades more than 200 journal papers have been published on this problem, e.g., see an online list of literatures [1]).

Because of the serial data dependence in the standard Gaussian elimination/factorization process prohibits any parallelization, different algorithms are designed which trade doing extra arithmetic operations for a higher degree of parallelism. Such trade-off is typical in designing parallel algorithms for problems whose efficient parallelization are not straight-forward. The solution of tri-diagonal system is such a problem that ideally shows the trade-off considerations in designing algorithms for not so straight-forward or hard to parallelize problems.

As we have mentioned that during the last four decades a large variety of parallel algorithms for the solution of tri-diagonal systems have been proposed, some well known examples are: the cyclic reduction (CR) algorithm [4,7], the cyclic elimination (CE), the recursive doubling algorithm [19], the block partitioned elimination algorithms [5,13,20], and the block portioned (Cholesky) factorization algorithms [9]. These and other algorithms are designed in an ingenious way and by different researchers in parallel computing through the years. The different approaches are often presented in a different way like partition of rows or columns of the matrix, index permutation and elimination tree, etc. In this section we use the graph framework to describe several known examples of parallel algorithms for tri-diagonal systems in a unified way.

3.1. *The cyclic reduction* (*CR*) *algorithm*

Let the dimension of the matrix be $n = 2^k - 1$. Consider a set of three consecutive equations, centered around $i = 2, 4, 6, 8, \ldots, n-2$. The basic idea of this type of algorithm [4] is to use the odd numbered equations, i.e., equations $(i-1)$ and $(i+1)$ to cancel the variable x_{i-1} and x_{i+1} in the i-th equation, resulting in $(\frac{n+1}{2} - 1)$ equations with only even numbered variables. This is described in the following.

$$a^{(0)}_{i-1,i-2}x_{i-2} + a^{(0)}_{i-1,i-1}x_{i-1} + a^{(0)}_{i-1,i}x_i = b^{(0)}_{i-1} \quad (1)$$

$$a^{(0)}_{i,i-1}x_{i-1} + a^{(0)}_{i,i}x_i + a^{(0)}_{i,i+1}x_{i+1} = b^{(0)}_i \quad (2)$$

$$a^{(0)}_{i+1,i}x_i + a^{(0)}_{i+1,i+1}x_{i+1} + a^{(0)}_{i+1,i+2}x_{i+2} = b^{(0)}_{i+1} \quad (3)$$

Multiplying Eqs. (1) and (3) with $-a_{i,i-1}/a_{i-1,i-1}$ and $-a_{i,i+1}/a_{i+1,i+1}$ respectively, and adding them to Eq. (2), we eliminate the variables x_{i-1} and x_{i+1} from Eq. (2), resulting in

$$a^{(1)}_{i,i-2}x_{i-2} + a^{(1)}_{i,i}x_i + a^{(1)}_{i,i+2}x_{i+2} = b^{(1)}_i \quad (4)$$

The CR algorithm consists of two phases: (I) the reduction phase, during which selective elimination of variables is done; and (II) the back-substitution phase, during which the values of the eliminated x_i's are recovered.

Using the framework, we can now generate the CR algorithm as shown in Fig. 7. The parallel elimination proceeds by starting with the elimination of all arcs initiated from even numbered nodes in the first step, followed by repeatedly eliminates the set of arcs initiated at even numbered nodes with a distance of 2^{k-1} to the end node at step k (they are independent from each other). Figure 8 illustrates the different elimination steps in the reduction- and back substitution phase of the reduction algorithm.

```
/* reduction phase */
FOR $m$ = 1 TO $log_2(n+1) - 1$ DO
   $h = 2^{m-1}$
   SELECT S(m) = $\{(2ih, 2ih - h), (2ih, 2ih + h) \mid$
         $i = 1, ..., (\frac{n+1}{2h} - 1)\}$
   for parallel elimination;
   ELIMINATE the arcs in $S(m)$ from the graph;
   UPDATE: add ll-arcs  $\{(2ih, 2ih - 2h),$
         $(2ih, 2ih + 2h) \mid i = 1, ..., (\frac{n+1}{2h} - 1)\}$;
END
/* back substitution phase */
FOR $m = log_2(n+1) - 1$ TO 1 DO
   $h = 2^{m-1}$;
   SELECT $S(2log_2(n+1) - m - 1) = \{(2ih + h, 2ih),$
         $(2ih - h, 2ih) \mid i = 1, ..., (\frac{n+1}{2h} - 1)\}$;
   ELIMINATE arcs in $S(2log_2(n+1) - m - 1)$ from graph;
   UPDATE: there are no ll-arcs;
END
```

Fig. 7. Algorithm 2 – The CR algorithm in terms of the framework.

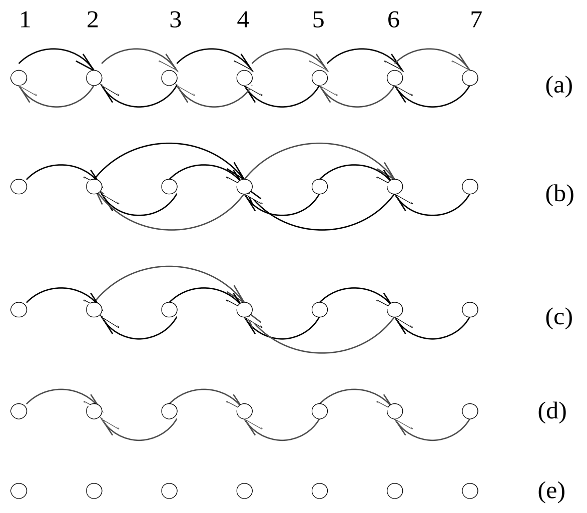

Fig. 8. Illustration of the graph transformation process corresponding to the CR algorithm. (b)-(c): the reduction phase; (d)-(e): the back-substitution phase.

3.2. *The cyclic elimination* (*CE*) *algorithm*

The CE algorithm repeatedly eliminates the arcs $(i, i+h)$ and $(i, i-h)$ with $h = 2^{m-1}$ at step m. This algorithm can be directly generated by the greedy algorithm (Fig. 3), the graphical illustration of the CE algorithm is shown in Fig. 5. The CE algorithm reduces the tridiagonal system to a diagonal system in $\log_2(n)$ elimination steps each consisting of 6 floating-point executions (assume that update of all non-zero coefficients are executed in parallel). So the parallel time complexity of the CE is $6\log_2(n) + 1$ (the last

```
/* reduction phase */
FOR $r = 1$ TO $log_l((n+1)/2)$ DO
    $h_1 = l^{r-1}$
    /* conform cyclic elimination */
    FOR $q = 1$ TO $m_1$ DO
        $h_2 = 2^{q-1}$;
    SELECT $S(q + (m_1 + 1)(r - 1)) =$
            $\{((i-1)lh_1 + jh_l, (i-1)lh_1 + jh_1 + h_1h_2),$
            $((i-1)lh_1 + jh_1 + h_1h_2, (\mathrm{i}-1)lh_1 + jh_1)$
            $| i = 1, ..., (n+1)/(lh_1),\ j = 1, ..., l - 1 - h_2\}$
            for parallel elimination;
    ELIMINATE arcs in $S(q + (m_1 + 1)(r - 1))$ from graph;
    UPDATE: add ll-arcs
            $\{((i-1)lh_1 + jh_1, (i-1)lh_1 + jh_1 + 2h_1h_2)$
            $| i = 1, ..., (n+1)/(lh_1),\ j = 1, ..., l - 1 - 2h_2\}$
            $\cup\{((i-1)lh_1 + jh_1 + h_1h_2, (i-1)lh_1 + jh_1 - h_1h_2)$
            $| i = 1, ..., (n+1)/(lh_1),\ j = 1 + h_2, ..., l - 1 - h_2\}$;
    END
    /* conform cyclic reduction */
    SELECT $S((m_1 + 1)r) = \{(ilh_1, ilh_1 - h_1),$
            $(ilh_1, ilh_1 + h_1)\ | i = 1, ..., (n+1)/(lh_1) - 1\}$
            for parallel elimination;
    ELIMINATE arcs in $S((m_1 + 1)r)$ from the graph;
    UPDATE: add ll-arcs   $\{(ilh_1, ilh_1 - lh_1),$
            $(ilh_1, ilh_1 + lh_1),\ | i = 1, ..., (n+1)/(lh_1) - 1\}$;
END
/* back substitution phase */
FOR $r = log_l((n+1)/2)$ TO 1 DO
    $v = l^{r-1}$;
    SELECT $S((m_1 + 2)log_l((n+1)/2) - r + 1) =$
            $\{(ilv + jv, ilv), (ilv - jv, ilv),\ | i = 1, ...,$
            $(n+1)/(lv) - 1, j = 1, ..., l - 1\}$
            for parallel elimination;
    ELIMINATE arcs $S((m_1 + 2)log_1((n+1)/2) - r + 1)$
            from the graph;
    UPDATE: there are no ll-arcs;
END
```

Fig. 9. Algorithm 3 – The ACER algorithm. $n = 2l^k - 1$, $l = 2, 3, \ldots, k = 1, 2, \ldots$, and $m_1 = \lceil \log_2(l-1) \rceil$.

division by the diagonal to obtain the solution vector takes 1 time unit). The time complexity is smaller than the CR algorithm which has a time complexity of $11 \log_2(n) - 11$. The price to be paid is the additional fill-arcs and thus a larger total number of floating point operations (when counted sequentially).

3.3. The recursive doubling algorithm

The CR and CE algorithms are based on the principle of Gaussian elimination, whereas the recursive doubling [19] and block Cholesky factorization algorithm in [9] are based on the principle of first factorizing the matrix A into a product of a lower triangular matrix and an upper triangular matrix with unit diagonals.

The recursive doubling algorithm computes the LU-factorization of a tridiagonal matrix. The computation of the factors is equivalent to a pair of linear recurrence equations (see e.g. [19]), and the computation of these linear recurrences can be performed in the same way as for the bidiagonal system discussed in the previous section (Fig. 4). The twisted factorization corresponds to the elimination of outgoing arcs from node 1 to node $n/2$, and incoming arcs from node n to node $n/2$. This algorithm can be generated by adding the constraints of no-fill during the parallel elimination/factorization to the selection operation.

4. The ACER algorithms

In this section we describe a class of new algorithms, called ACER (Alternating Cyclic Elimination and Reduction algorithm) [12], which combines the advantages of the well known CE and CR algorithm.

As we have observed in the previous sections, the CE algorithm uses a greedy approach in eliminating the arcs in parallel which has a smaller time complexity of $4 \log_2(n) + 1$ compared to $8 \log_2(n+1) - 8$ for the CR algorithm. When we look at the total number of floating point operations or the number of fill-arcs, we see that the number of fill-arcs for the CR algorithm is $2(n-1) - 4 \log_2(\frac{n+1}{2})$ which is much smaller than $2n \cdot \log_2(n) - 4n + 4$ for the CE algorithm. This brings us to the idea of combining the advantages of these two algorithms.

Using the graph framework we can easily visualize and study the different variants of parallel elimination algorithms, and that results in an ACER algorithm which performs alternating CR and CE until the entire matrix system is solved. Like the CR algorithm, the ACER algorithm comprises a reduction and a back-substitution phase, each consisting of k parallel elimination steps. However, unlike the CR algorithm, each of these k steps in the reduction phase again consists of one or more sub-steps: each step consists of zero or more CE sub-steps, followed by one single CR sub-step at the end. In following the two phases will be described in detail using the graph notations. But before doing so, first some notations of sets of nodes or arcs are introduced.

Figure 9 describes the ACER algorithm in terms of graph transformations using the framework.

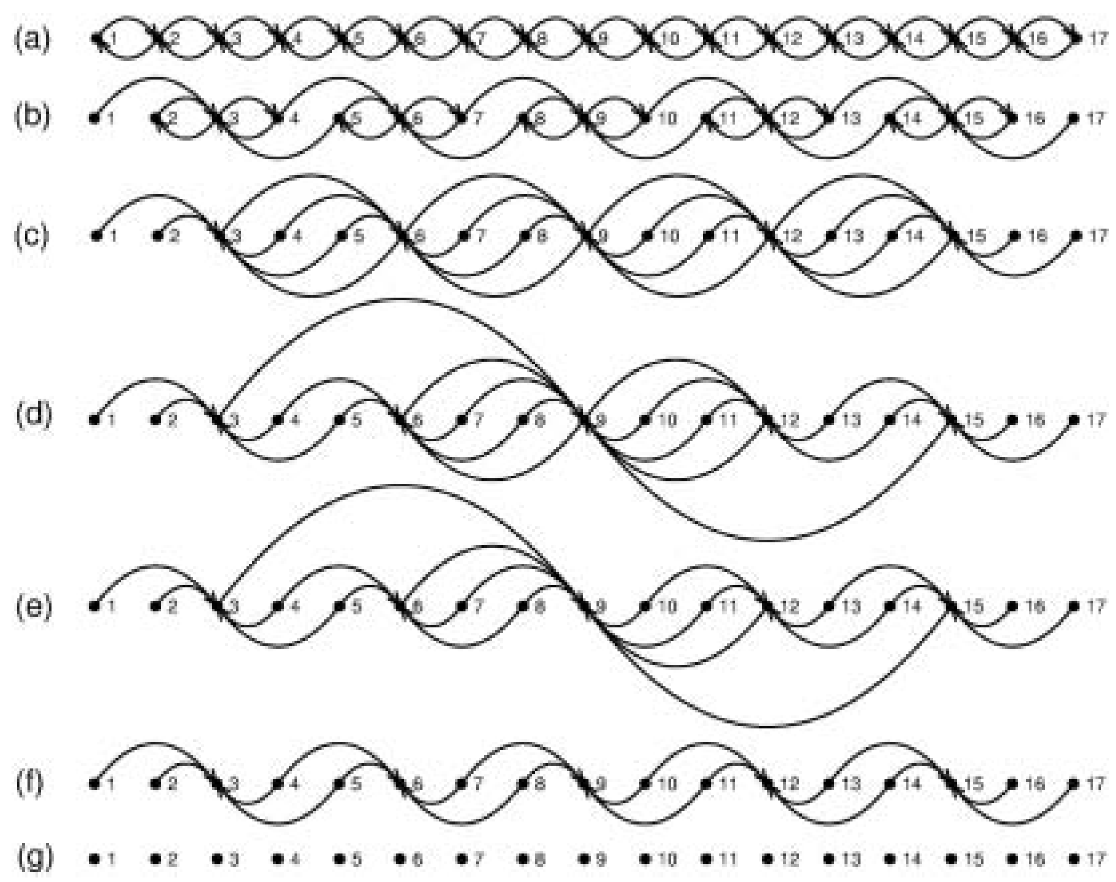

Fig. 10. Illustration of the ACER-3 algorithm ($l = 3$) for $n = 17$ ($k = 2$). (a)→(e): the reduction phase; (e)→(g): the back-substitution phase.

The ACER algorithm in Fig. 9 contains $\log_l \frac{(n+1)}{2}$ reduction phases, l is the basic partition size of the matrix where the cyclic elimination process is applied. Within each reduction phase t, cyclic eliminations are performed for the partitions (sub-matrices) of $h_1 = l^{t-1}$ rows, followed by a single reduction step at the end of each phase. Figure 10 shows an example of ACER algorithm $l = 3$ for $n = 17$ and $k = 2$. Notice that the ACER algorithm is flexible in taking the values of the matrix dimension n, which can be $n = 2l^k - 1$ with $l = 2, 3, \dots$ and $k = 1, 2, 3, \dots$. This is in contrast to the limited values for both the CE ($n = 2^k$) and CR ($n = 2^k - 1$) algorithms. By choosing different base l, we can obtain a variety of variants of ACER algorithms. Notice that when $l = 2$, this special case of the ACER algorithm is identical as the CE algorithm.

The time complexity of the ACER algorithm[3] is $(4m_1 + 8 - 1_{[m_1>0]}) \log_l(\frac{n+1}{2})$ and the number of fill-ins is $\{2m_1 - \frac{2^{m_1+1}-4}{l-1}\}(n-1) - 2l \cdot \log_l(\frac{n+1}{2})$, where $m_1 = \lceil \log_2(l-1) \rceil$ and $l_{[m_1>0]}$ is an indicator function, $1_{[m_1>0]} = 1$ if $m_1 > 0$; =0 otherwise. Take for example $l = 3$, then the time complexity of the ACER algorithm is approximately $6.9 \log_2(n+1)$ which is larger than $4\log_2(n) + 1$ for the CE algorithm but smaller than the time of $8 \log_2(\frac{n+1}{2})$ for the CR algorithm. Comparing the number of fill-ins, the ACER algorithm has approximately $2n + O(\log_l(n))$ fill-ins which is a factor of $\log_2(n)$ smaller than the CE algorithm and about the same as the CR algorithm. Thus the ACER algorithms combine the advantages of the CE and CR algorithms.

[3]If the update of a coefficient is calculated by only one processor (instead of maximum parallelism)then the time complexity of the ACER algorithm becomes $(6m_1 + 11 - 3 \cdot 1_{[m_1>0]}) \log_l(\frac{n+1}{2})$.

5. Concluding remarks

We have presented a general graph theoretic framework which unifies many different parallel algorithms for sparse matrix computations. The most significant of this framework based on graph transformation is that it is able to introduce parallelism beyond the constraint of the usual data dependence analysis. The framework uses directed graphs instead of the traditional undirected graphs which are not capable of describing many known parallel algorithms. The results regarding the property of parallel elimination of multiple arcs and the test on the update conflict now enable us to design different variants of parallel algorithms. An important application of the framework is that it provides us with a simple mechanism to optimize and modify the known parallel algorithms. This makes not only the task of manually designing a parallel algorithm simpler, but it also provides us with a means to automatically generate and optimize parallel algorithms.

Furthermore, although we have limited our discussions to the example of a tri-diagonal matrix, the framework can be easily extended to the other type of sparse matrix computations. The theorems and lemmas are also valid for general sparse matrices. The power of the graph formalism lies in that we design parallel algorithms beyond the ability of detecting parallelism in an existing algorithm. We believe that similar frameworks can be defined for sparse matrix computations other than the Gaussian-elimination and factorization. We are currently studying the problems of sparse eigenvalue algorithms through orthogalization procedures. Different type of computations may have different graph representations, for instance, elimination of an arc has different meaning for Gaussian elimination and for Givens transformation. Therefore extending the framework to other sparse linear algebra problems comprises defining the problem as a corresponding graph transformation problem. A number of problems needs to be further investigated. For example, how to apply the graph framework in compliers for automatically generating and optimizing parallel programs. This will give parallel compilers the ability to introduce parallelism which is not possible with the existing transformation techniques on programs. Another topic is to apply framework to analyze and minimize the communication requirement.

References

[1] A Bibliography on Parallel Solution of Tri-diagonal Systems of Equations. http://ta.twi.tudelft.nl/wagm/users/lin/Biblio/tri_sol.html.

[2] I.S. Duff, A.M. Erisman and J.K. Reid, *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford, 1986.

[3] A. George and J.W.H. Liu, *Computer solution of large sparse positive definite systems*, Prentice-Hall, 1981.

[4] R.W. Hockney, A fast direct solution of Poisson's equation using Fourier analysis, *Journal of ACM* **12** (1965), 95–113.

[5] S.L. Johnsson, Solving tridiagonal systems on ensemble architectures, *SIAM J. Sci. Stat. Comput.* **8** (1987), 354–392.

[6] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure and M. Wolfe, Dependence graphs and complier optimizations, *Proc. of the 8th ACM Symp. Principles of Programming Languages*, Williamsburg, VA, 1981.

[7] J.J. Lambiotte and R.G. Voigt, The solution of tridiagonal linear systems on the CDC STAR-100 computer, *ACM TOMS* **1** (1975), 308–329.

[8] H.X. Lin, *A methodology for the parallel direct solution of finite element systems*, Ph.D. thesis, Delft University of Technology, 1993.

[9] H.X. Lin and M.R.T. Roest, Parallel solution of symmetric banded systems, in: *Parallel Computing: Trends and Applications*, G.R. Joubert, D. Trystram, F.J. Peters and D.J. Evans, eds, Elsevier Science, 1994, pp. 537–540.

[10] H.X. Lin, A Unifying Graph Model for Designing Parallel Algorithms For Tridiagonal Systems, *Parallel Computing* **27** (2001), 925–939.

[11] H.X. Lin, Designing parallel sparse matrix algorithms beyond data dependence analysis, *Proc. ICCP 2001, Workshop High Performance Scientific Engineering Computing with Applications* (Keynote), IEEE Computer Society Press, Valencia, Sept 3–7, 2001, pp. 7–13.

[12] H.X. Lin and J. Verkaik, ACER: alternating cyclic elimination and reduction algorithm for the solution of tri-diagonal systems, *Proc. of 2002 Int'l Symp. Distributed Computing and Applications*, Q.P. Guo et al., eds, Wuxi, China, December 16–18, 2002, pp. 17–21.

[13] U. Meijer, A parallel partition method for solving banded systems of linear equations, *Parallel Computing* **2** (1985), 33–43.

[14] M.F.P. O'Boyle and P.M.W. Knijnenburg, Integrating loop and data transformations for global optimization, *Journal of Parallel and Distributed Computing* **62** (2002), 563–590.

[15] S. Parter, The use of linear graphs in Gaussian elimination, *SIAM Review* **3** (1961), 119–130.

[16] F.J. Peters, Parallel pivoting algorithms for sparse symmetric matrices, *Parallel Computing* **1** (1984), 99–110.

[17] P. Petersen and D. Padua, Static and dynamic evaluation of data dependence analysis techniques, *IEEE Trans. Par. & Distr. Systems* **7** (1996), 1121–1132.

[18] D.J. Rose, A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations, in: *Graph Theory and Computing*, R.C. Read, ed., Academic Press, New York, 1972, pp. 184–218.

[19] H.S. Stone, An efficient parallel algorithm for the solution of a tridiagonal linear system of equations, *Journal of ACM* **20** (1973), 27–38.

[20] H.H. Wang, A parallel method for tridiagonal equations, *ACM TOMS* **7** (1981), 167–183.