A Near-optimal Real-time Hardware Scheduler for Large Cardinality Crossbar Switches

Raymond R. Hoare* Concurrent EDA, LLC

Zhu Ding† Union Switch & Signal

Alex K. Jones‡ University of Pittsburgh

Abstract

The maximum matching algorithm for bipartite graphs can be used to provide optimal scheduling for crossbar based interconnection networks. Unfortunately, maximum matching requires $O(N^3)$ time for an $N \times N$ communication system, which has limited its application to real-time network scheduling. In this paper, we show how maximum matching can be reformulated in terms of Boolean operations rather than the more traditional formulations. By taking advantage of the inherent parallelism available in custom hardware design, we introduce three Maximum Matching implementations in hardware and show how we can trade design complexity for performance. Specifically, we examine a Pure Logic Scheduler with three dimensions of parallelism, a Matrix Scheduler with two dimensions of parallelism and a Vector Scheduler with one dimension of parallelism. These designs reduce the algorithmic time complexity down to $O(1)$, $O(K)$, and $O(KN)$, respectively, where K is the number of optimization steps. While an optimal scheduling algorithm requires $K = 2N - 1$ steps, our simulation results show that the scheduler can achieve 99% of the optimal schedule when $K = 9$. We examine hardware and time complexity of these architectures for crossbar sizes of up to N = 1024. Using FPGA synthesis results, we show that a greedy schedule for various sized crossbars, ranging from 8×8 to 256×256, can be optimized in less than 20 ns per optimization step. For crossbars reaching 1024×1024 the scheduling can be completed in approximately 10μs with current technology and could reach under 90 ns with future technologies.

1 Introduction

Maximum matching in bipartite graphs is a widely studied problem [1–4]. Maximum matching provides the optimal number of links between a set of sources and a set of destinations based on the requirements of system. A crossbar interconnect can be represented as a bipartite graph where each vertex corresponds to a communication node and each edge represents a link between the two nodes. Because bipartite maximum matching is optimal, the maximum network utilization is achieved. However, due to its $O(N^3)$ operational complexity, the running time of this technique is not scalable.

Developments of electrical, analog, and optical network technologies continue to push the limits of network switches. It is now possible to place large crossbars within single chips and even larger crossbars within communication racks, as seen in the IBM BlueGene/L [5] and NEC Earth Simulator [6]. Scheduling these switches to achieve maximum throughput in the system is increasingly challenging. Currently, many of these systems use a greedy scheduling approach that results in a fast schedule, but achieves considerably sub-optimal results. This leads to network underutilization and degradations in communication time. If a maximum matching approach could be used in place of a greedy scheduler, it would improve network utilization and reduce communication time.

One of the benefits of the maximum matching algorithm is that it continuously improves an existing matching until optimality is reached. By first utilizing a heuristic such as the greedy strategy a baseline matching can be established. Then, the maximum matching can discover improvements, which can be used to further improve the resulting solution until either a fixed amount of time has expired or the best solution has been reached.

This paper describes three maximum matching architectures. By utilizing fundamental properties of hardware logic it is possible to detect potential paths for optimization in parallel and trace back the result. As a result of implementing a Pure Logic Scheduler with three dimensions of parallelism, a Matrix Scheduler with two dimensions of parallelism, and a Vector Scheduler with one dimension of parallelism, the maximum matching algorithmic run-time complexity can be reduced from $O(KN^2)$ to $O(1)$, $O(K)$ and $O(KN)$, respectively, where K is the number of optimization steps for an $N \times N$ crossbar. The bipartite algorithm improves results generated by a greedy algorithm [7] and is able to grant more requests. By examining our simulation results, we find that near-optimal schedules (approximately 99% of optimal) can be established in no more than nine optimization steps

*e-mail: hoare@concurrenteda.com

†e-mail:zhuding@connecttime.net

‡e-mail:akjones@ece.pitt.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC2006 November 2006, Tampa, Florida, USA

0-7695-2700-0/06 $20.00 ©2006 IEEE

($K = 9$).

The rest of the paper is organized as follows. Related work on maximum matching algorithms is introduced in Section 2. Section 3 introduces some general terms and theorems. Section 4 describes the hardware algorithms and architectures that solve the maximum matching problem. In particular, the tradeoff of hardware timing and area cost are considered. System simulation is described with the results in Section 6. Finally, we discuss conclusions and future research directions in Section 7.

2 Prior work

The maximum matching algorithm allows 100% throughput under uniform traffic for packet switching networks [8]. Several researchers have applied bipartite maximum matching to solve network switch configuration [9, 10]. Weller and Hajek [9] applied a batch scheduling extension to maximum matching to solve the fairness problem for a crossbar network called critical maximum matching. Maximum matching can also be used to determine and utilize the optimal multiplexing degree for time-division multiplexing (TDM) networks [10]. While maximum matching provides extremely attractive network characteristics, the algorithmic complexity makes it impractical for large switches.

The sequential maximum algorithm was first proposed and proved optimal by Edmonds [11]. It iteratively searches for augmenting paths and increases the cardinality of the matching set. This has been shown to require $O(N^3)$ time with a maximum augmenting path length of $2N-1$.

Subsequently, most sequential algorithms have been derived from the same concept [12]. Additionally, several parallel algorithms have been proposed to solve this problem. Mulmuley et. al. proposed a parallel version of the algorithm based on the pugilistic lemma requiring $O(N^{3.5})$ processors in parallel [4]. Hanckowiak et. al. implemented a parallel algorithm with computation complexity of $O(\lg^6 N)$ for approximating the maximum matching with the matching degree at least 2/3 of the maximum matching degree [13, 14].

To make the maximum matching algorithm practical for a scheduler within a switch, the algorithm must execute quickly *and* be able to fit within a single silicon device. Prior efforts in parallelizing the maximum matching algorithm utilize multiple processors that execute sequential instructions. In contrast, the underlying algorithm is based on sets, which can be represented as binary bits. Thus, the algorithm can be implemented as a set of operates acting in parallel on these bits. Our approach utilizes parallelism available in a custom hardware solution.

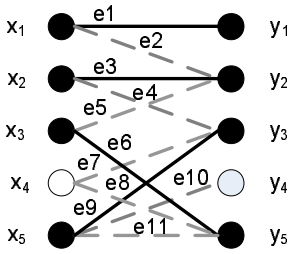


Figure 1: A bipartite graph representing a crossbar schedule.

3 Background

In this section, we introduce general terminology and fundamental theorems related to the optimal maximum matching algorithm that will be used in its description. More detailed definitions and proofs are found in [11]. We utilize these fundamental principles to create an optimizing scheduler.

Definition 1 *Bipartite graph*

A undirected graph, G, consisting of vertices and edges, $G = (V,E)$, is a Bipartite Graph if G can be separated into two sets V_1 and V_2, such that $V_1 \cap V_2 = \theta$ and $V_1 \cup V_2 = V$ and for any edge $e = (v_i, v_j) \in E$, then $v_i \in V_1, v_j \in V_2$.

Figure 1 gives a example of bipartite graph. $V_1 = \{x_1, x_2, x_3, x_4, x_5\}$,
$V_2 = \{y_1, y_2, y_3, y_4, y_5\}$, $E = \{e_1, e_2, e_3, \ldots, e_{11}\}$.

To configure a crossbar, it is necessary to schedule communications between pairs of nodes such that each node is only involved in sending a single message at any given time. In this case, the edges of the graph represent a communication connection between a sender and receiver. In this graph notation, each node is represented as an (x, y) pair that represents the sending and receiving node.

Definition 2 *Maximum matching*

In bipartite graph $G = (V,E)$, for an edges set M, $M \subset E$, and there does not exist an M' such that $|M'| > |M|$ and $M' \subset E$, then M represents a maximum matching.

An objective of crossbar scheduling is to maximally utilize the routing resources available in the crossbar. We examine and optimize the scheduling of the crossbar at a particular instance in time, based on a set of requests in the system. These requests are represented as edges in the bipartite graph. Consequently, the largest possible set of edges are scheduled at each time instance.

Definition 3 *Saturated and unsaturated edges*

Given a particular matching $M \subset E$, all edges, $e \in M$ are considered to be *saturated*. All edges $e' \in E$ but $e' \notin M$ are considered to be *unsaturated*.

For crossbar scheduling, we start with a configuration and apply the maximum matching algorithm to improve the bandwidth utilization of the crossbar. In this context, an established configuration between nodes X and Y is said to be saturated. A communication request that is not granted is said to be unsaturated.

Definition 4 *Saturated and Unsaturated Vertices*

Given a particular matching $M \subset E$, all vertices that are part of a saturated edge are considered to be saturated. Conversely, all vertices that are not part of a saturated edge are considered to be unsaturated.

Definition 5 *Augmenting Path*

If a path P in G is constructed by alternating unsaturated edges and saturated edges, and both ends of the path are unsaturated, then path P is called *augmenting path*.

Theorem 1 *M is a maximum matching of G, iff G does not contain an augmenting path within M.*

Corollary 1 *If M is not a maximum matching of G, then there exists a augmenting path within M.*

Corollary 2 *Given a matching M of G and an augmenting path P, within M, then there exists a matching M' that does not contain P whose cardinality is larger than M.*

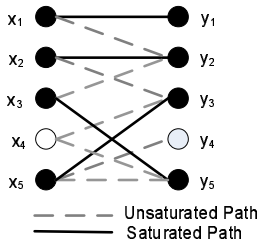
Theorem 2 *Given a matching M of G and an augmenting path P, a new matching M' can be created from M such that $|M'| > |M|$ by removing all saturated edges in P from M and by adding all unsaturated edges in P to G.*

For example, Figure 1 shows a matching set using the darkened edges. The augmenting path in this graph contains edges $\{e_7, e_9, e_{10}\}$. By removing the saturated edge e_9, the unsaturated edges e_7 and e_{10} can be added; thereby increasing the number of edges in the matching set.

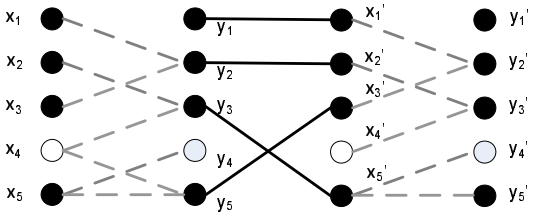
In our hardware implementation, augmenting paths may be searched in parallel using a customized architecture. All computation is based on pure hardware logic, matrix, or vector operations. However, asymptotic notation breaks down in this case as our architecture is comprised of $O(N^2)$ gates within a single "processor" rather than $O(N^2)$ processors. Essentially, our technique exploits the transistor density possible in current hardware fabrication techniques to achieve superior performance while fitting within a single device.

4 Specialized processors for optimal scheduling

In this section, a parallel maximum matching algorithm is presented that can be realized with the inherent performance and density of hardware logic. All possible augmenting paths are detected in parallel. Then one path is traced back



(a) Original bipartite graph with matching M



(b) Unfolded bipartite graph with matching M

Figure 2: Original and unfolded bipartite graphs.

for the exact optimization. Using the algorithm, we propose three hardware designs: (1) Pure Logic Architecture, (2) Matrix Architecture, and (3) Vector Architecture. The hardware timing and area analysis shows that for 90nm FPGA technology, the Pure Logic Architecture is feasible for small systems (16 or fewer nodes), the Matrix Architecture is appropriate for large systems (32 – 128 nodes), and the Vector Architecture can handle very large systems (128 – 1024 nodes).

4.1 Maximum matching algorithm

To extract the available parallelism in the maximum matching algorithm, the Bipartite Graph is unfolded as shown in Figure 2. The graph starts with an unsaturated path and then adds multiple pairs of saturated followed by unsaturated paths.

Consider the original Bipartite Graph from Figure 2(a). The source nodes on the left (X) are labeled x_1 through x_5 and destination nodes on the right (Y) are labeled y_1 through y_5. Requested communications are shown as connections between nodes in X and Y. Darkened connections represent an existing schedule and darkened nodes represent nodes that are involved in a communication. These scheduled connections and nodes are labeled as *saturated* while the dashed connections and nodes are labeled as *unsaturated*.

The original maximum matching algorithm uses sets to describe the algorithm and can be found in [12]. Conceptually, the algorithm removes K saturated links (i.e., scheduled) and

adds $K+1$ links. This increases the number of scheduled communications in the final schedule by 1. In Figure 2(a), it can be seen that by removing $\{x_5, y_3\}$, $\{x_4, y_3\}$ and $\{x_5, y_4\}$ can be added.

The algorithm for finding this optimization involves finding an augmenting path, or an alternating sequence of unsaturated and saturated links between X and Y. The starting and ending nodes must be unsaturated. In the example in Figure 1, the path is $x_4 \rightarrow y_3 \rightarrow x_5 \rightarrow y_4$. The length of the path does not matter but a particular vertex can only be included once.

In our representation of this algorithm, we split the graph into two graphs, labeled saturated and unsaturated. The unsaturated graph contains only unsaturated paths and the saturated graph contains only saturated paths. For the saturated path graph, we place the Y nodes on the left and the X nodes on the right. We then create a new graph by placing sequences of unsaturated then saturated graphs next to each other with overlapping vertices. Finally, an unsaturated graph is added to the far right of the graph. In essence, we flip the original graph over the Y axis an even number of times and only keep the unsaturated paths in the odd stages and only keep the saturated paths in the even stages. The optimization problem is reduced to finding a path from left to right through the graph starting and ending with an unsaturated vertex.

Figure 2(b) shows the new unfolded graph with three stages. We use x'_i and y'_i to represent the unfolded vertex of x_i and y_i respectively, where $0 \leq i < N$. From this new graph, two legitimate paths existing through stages 1 and 2, namely $x_4 \rightarrow y_3 \rightarrow x'_5$ and $x_4 \rightarrow y_5 \rightarrow x'_3$, but only one of the paths traverses stage three to end at an unsaturated node, namely $x_4 \rightarrow y_3 \rightarrow x'_5 \rightarrow y'_4$. This path is equivalent to the path, $x_4 \rightarrow y_3 \rightarrow x_5 \rightarrow y_4$, as shown in the Figure 1.

The optimization to the schedule removes the saturated links that exist in the final path and adds the unsaturated links. In this algorithm, there is always one more unsaturated link than saturated and thus, the result is always a schedule that contains one additional link.

To determine if *any* optimizations can be found from a particular unfolded graph, we can conceptually treat the graph as if it was physically wired system and attach a voltage to the unsaturated nodes at the far left of the graph and light bulbs to the unsaturated nodes at the far right side of the graph. If any optimization path exists, then one of the light bulbs will illuminate. Thus, it can be seen that a single left-to-right pass through the graph is sufficient to determine if there are optimizations available.

There are three parts to the hardware algorithm, namely: (1) detection of augmenting paths, (2) isolation of a single augmenting path and (3) matching set update. Path detection is the most complex as it requires all possible paths to be checked. This is performed in parallel to discover all paths. The second pass backtracks through the graph and isolates a single path. The third step is to update the schedule (i.e., the Matching Set).

The variables used are defined as follows

- The proposed parallel maximum matching algorithm takes an iterative approach and searches for augmenting paths of increasing length. Let K be the number of steps in the original algorithm and is the maximum length of the augmenting path.
- N is the system size of an $N \times N$ crossbar.
- F$[k]$ is a matrix representing the forward path of step k.
- B$[k]$ is a matrix representing the backtrack path of step k.
- UnsaturatedX$[i, j]$ is the mapping of unsaturated vertex x_i in X to vertex y_j in Y.
- SaturatedY$[i, j]$ is the mapping of saturated vertex y_i in Y to vertex x_j in X.
- XMask$[k]$ and YMask$[k]$ are bit-vectors of size N in step k.

Figure 3 shows the procedure of detecting augmenting paths. We use XMask$[k]$ to represent the status of nodes at step k. If x_i is unsaturated, the i^{th} bit in XMask$[k]$ is set to '1'; otherwise, it is set to '0'. In this example, only x_4 is unsaturated after greedy scheduling, therefore XMask$[1]$ is {0 0 0 0 1 0}. We use XMask$[1]$ to filter UnsaturatedX. If the i^{th} bit of XMask is '1', the i^{th} row of Unsaturated X is enabled. In Figure 3, only the 4^{th} bit in XMask is '1', therefore only the 4^{th} row of UnsaturatedX is enabled. This is equivalent to point-wise vector matrix multiplication. Because all values are either '0' or '1' the multiplication can be completed with a logical AND operation. The resulting matrix is F$[1]$. XMask$[2]$ is constructed by column-wise OR reduce of F$[1]$. Similarly, after filtering SaturatedY using XMask[2], we get F$[2]$. By applying column-wise OR reduce on F$[2]$, we get XMask$[3]$. Finally, after filtering UnsaturatedX with XMask$[3]$ to generate F$[4]$, a column-wise OR reduce of F$[4]$ results in XMask$[4]$.

We define a vertex as active if it is traversed during discovery of augmenting paths. y_3 and y_5 are active after first step, shown by a '1' in XMask$[2]$. x'_3 and x'_5 are active in the second step and y'_2, y'_4, and y'_5 in the final step, shown by XMask$[3]$ and XMask$[4]$, respectively. Finally, only y'_4 is selected because of the three vertices identified, this is the only unsaturated vertex. The next step in the algorithm is to isolate the augmenting path and update the matrices. The optimization then begins again with the new matrices.

Once one or more augmenting paths have been found, exactly one of these paths must be identified for optimization. This requires backtracking through the F matrices and selecting a single path to isolate along the way. It is

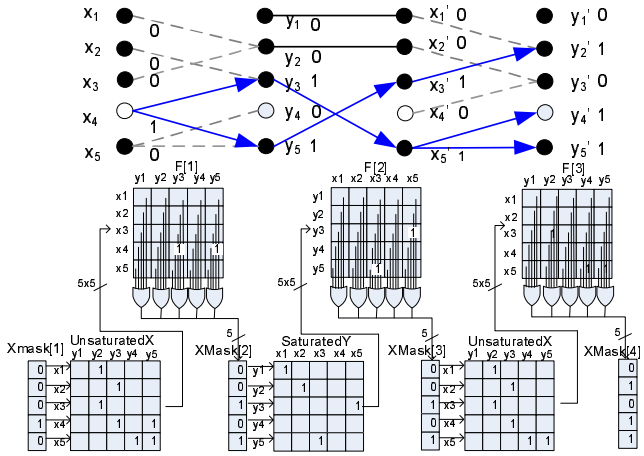


Figure 3: Parallel tracing of potential augmenting paths as described in detection of augmenting paths algorithm

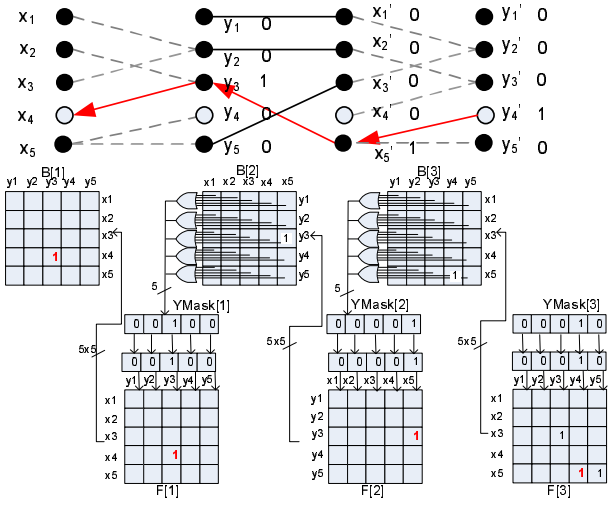


Figure 4: Isolation of a single path within the augmenting paths.

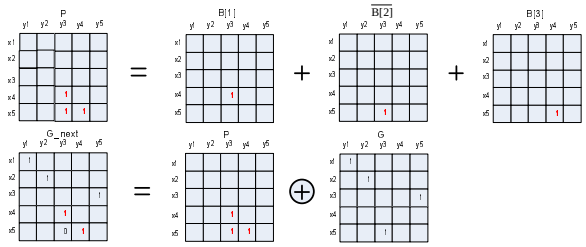


Figure 5: Matching set update.

a fairly straightforward extension to isolate multiple paths non-overlapping paths in parallel but for simplicity, we only isolate a single path, which is shown in Figure 4. Isolating multiple backwards paths does not improve the runtime complexity of the algorithm as in the worst case, all paths could overlap. We use YMask to represent the status of nodes.

We start from the rightmost side for path isolation. Because y'_4 is unsaturated, the 4^{th} we set this bit in YMask[3] to '1'. If there were multiple unsaturated vertices in XMask[4], we pick the lowest possible bit for YMask[3]. We use YMask[3] to filter F[3] and we call the result B[3]. YMask[2] is the row-wise reduce OR operation of B[3]. YMask[2] is used to filter F[2] and generate B[2]. Following the same approach, B[1] is generated. Visually, x'_5 is active after first step. y_3 is active after the second step and x_4 is active after the final step. The augmenting path is isolated, which is $y'_4 \rightarrow x'_5 \rightarrow y_3 \rightarrow x_4$. It is equivalent to the path $y_4 \rightarrow x_5 \rightarrow y_3 \rightarrow x_4$ in the original graph.

The final step after isolating a single augmenting path is to update the matrices. The updating of the connection matrices is illustrated in Figure 5. All unsaturated edges in the augmenting path are added to the matching set and all saturated edges in the augmenting path are removed from the matching set. Thus, all odd steps in the backtracking algorithms B[1, 3, 5, ..] are added while the even steps are removed. We first generate and store the augmenting path, which is a matrix P. P is generated by adding all odd B matrices and the transpose of all B matrices. In this example, this is the equivalent of applying OR logic on B1, B2 transpose, and B3. Then the optimized matching is achieved by applying XOR logic on P and the original matching matrix G.

Our proposed algorithm is summarized in Figures 6 and 7. The pseudo code is described in a general form that can be implemented by pure hardware logic, matrix, or vector operations. We describe the tradeoffs and performance of each of these three options in the following section.

The detection of augmenting paths algorithm shown in Figure 6 moves left to right through the unfolded graph in parallel. Line 2 creates a vector that contains unsaturated vertices. The UnsaturatedX matrix is a connection matrix and by using the XMask, we remove all paths that do not originate from an unsaturated vertex. Line 4 determines which ver-

```
    // Start from step 1
1.  i = 1;
2.  XMask[1] = UnsaturatedXVertices;
3.  F[1] = XMask[1]*UnsaturatedX;
4.  XMask[2] = ColumnwiseReduceOR(F[1]);
    // See if any augmenting path is found
5.  PathFound =
      ReduceOr (XMask[2]*UnsaturatedYVertices);
6.  while Not PathFound and i ≤ K {
      //Follow a saturated path
7.    F[i+1] = Xmask[i+1]*SaturatedY;
8.    XMask[i+2] =
        ColumnwiseReduceOR(F[i+1]);
      // Follow an unsaturated path
9.    F[i+2] = XMask[i+2]*UnsaturatedX;
10.   XMask[i+3] =
        ColumnwiseReduceOR(F[i+2]);
      // See if any augmenting path is found
11.   PathFound = ReduceOr (XMask[i+3]*
        UnsaturatedYVertices);
12. }
```

Figure 6: Detection of augmenting paths algorithm.

```
15. S = Final step number
    // Select one of the active unsaturated vertices
16. i = S;
17. YMask[i] = SelectOne(XMask[S]);
18. for (i = S; i > 1; i − −) {
19.       B[i]=YMask[i]*F[i];
20.       YMask[i]=
              SelectOne( RowwiseReduceOR(B[i]));
21. }
```

Figure 7: Isolation of a single augmenting path.

tices in second column are active, that is, those vertices that are part of a path. This is performed by a column-wise OR reduce of the matrix. These three lines represent a single step in the *unfolded* graph.

The loop beginning on line 6 represents two steps in the unfolded graph, the first being traversing saturated edges and the second being traversing unsaturated edges. This loop can iterate multiple times as the graph can be unfolded multiple times.

The isolation of a single augmenting path algorithm shown in Figure 7 moves right to left through the unfolded graph in parallel. Line 17 selects the first unsaturated vertex found in the augmenting paths detection to begin the process and the loop starting in 18 creates the appropriate B matrices and YMasks along the way.

From the code and examples, it can be seen that the operations required for the maximum matching algorithm are fundamentally vector are matrix operations. We further observe that the data types required for these operations are all single bit Boolean for which a full blown ALU is overkill. However, traditional processors do not perform matrix operations well, even when they are simple and Boolean. Thus, we will next explore three alternative architectures for acceleration of these algorithms.

Table 1: Complexity analysis of three maximum matching architectures. N presents the number of nodes in the system and K represents the number of optimization steps performed.

	Architecture		
	Pure logic	Matrix	Vector
Wire complexity	$O(KN^2)$	$O(N^2)$	$O(N)$
Logic complexity	$O(KN^2)$	$O(N^2)$	$O(K)$
Memory complexity	0	$O(KN^2)$	$O(KN^2)$
Latency in cycles	$O(1)$	$O(K)$	$O(KN)$
Multiple iterations	Disable	Enable	Enable

5 Hardware timing and area cost

The maximum matching algorithm can be processed in a single clock cycle using pure combinational logic or in multiple cycles by using either matrix or vector operations. We first examine the asymptotic tradeoffs and then describe each of three architectures.

For small values of N, a purely hardware version of the maximum matching can be performed in a single cycle and called the pure logic scheduler. The matrix scheduler performs each of the steps in the augmenting path detection and isolation algorithms in a single cycle. This requires a register file for storing the intermediary results. The vector scheduler is focused on large values of N where it is not possible to have an $O(N^2)$ register file. This architecture breaks each matrix operation up into vector operations and uses traditional memories rather than registers for storage.

According to the hardware structure, the hardware cost can be computed approximately as shown in Table 1, where N represents the number of nodes and K represents the number of optimization steps.

The augmenting path detection phase propagates signals forward through the unrolled graph. It requires logic computations such as OR and AND. Likewise the path isolation propagates signals back through the unrolled graph and uses similar logic as path detection. A priority selector is used to select a particular starting point for isolation, which adds an additional slight delay.

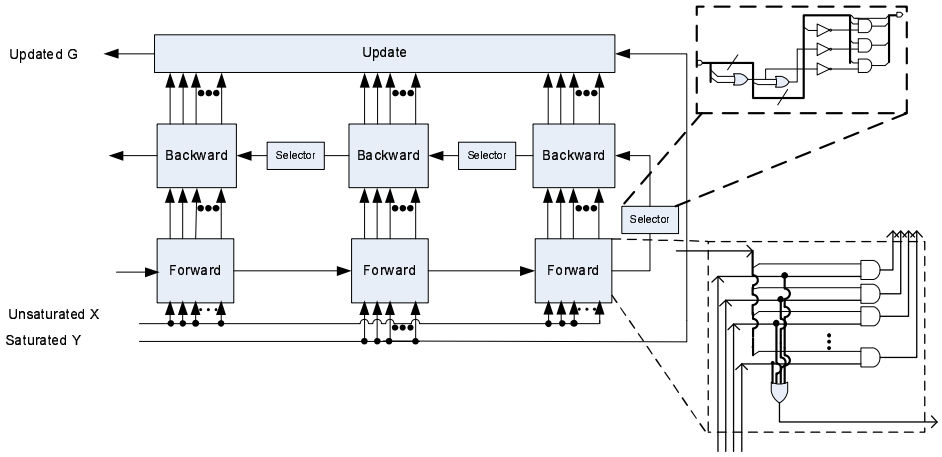


Figure 8: Pure logic scheduler to implement the maximum matching algorithm.

5.1 Pure logic scheduler

In the Pure Logic Scheduler, the maximum matching optimization phase is performed by pure combinational logic, as shown in Figure 8. A sequence of forward logic blocks is responsible for detection of the augmenting paths while a sequence of backward logic blocks is responsible for isolation of a single augmenting path. Both the forward and backward logic blocks are combined together to form a single block of hardware. This consumes a large amount of logic and routing resources. However, once signals propagate the circuit, the optimization result is achieved. Therefore, for relatively small number of optimization steps and graph sizes, this approach is feasible.

5.2 Matrix scheduler

The Matrix Scheduler breaks the algorithms down into matrix operations. Each of the steps in the algorithms is performed by a Matrix ALU in a single cycle. Matrices are stored in a register array. During each cycle, one matrix is read from the register array. The mask bits enable outputs of each row in the original matrix by N OR gates. Each of the OR gates is $N+1$-bits wide. ReduceOR is performed on the outputs using N N-bit OR gates, which generate an N-bit signal R, as shown in Figure 9. R is XMask vector when the Matrix ALU operates for detection augmenting paths. When performing isolation step, R is filtered by a selector to generate the YMask vector.

With the Matrix ALU, the original algorithm is implemented as matrix operations. Let us use MatrixALU( ) to represent the function performed by the Matrix ALU. For example, the code from line 4 to line 12 in Figure 6 is written as shown in Figure 10 by using MatrixALU( ).

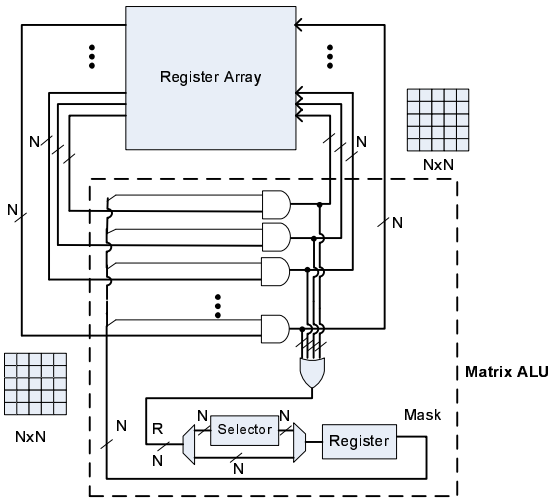


Figure 9: Matrix scheduler for the maximum matching algorithm.

This approach has better scalability than the pure logic approach and only requires $O(N^2)$ hardware. For relatively moderately sized N, this is still reasonable.

5.3 Vector Scheduler

The Vector Scheduler implements N-bit wide memory reads and vector operations in a single cycle. This architecture utilizes traditional memory components. Figure 11 shows a block diagram of the Vector Scheduler. During each cycle, a vector of a matrix is read from the memory. The output of the vector is enabled by one bit of a Mask. The bit is retrieved from XMask and YMask, for path detection and path isolation, respectively. One mask bit from a Mask vector is output during each cycle. The approach for generating XMask and

```
4.  while Not PathFound and i < K {
        // Follow an saturated path
5.      (F[i+1], XMask[i+2]) =
            MatrixALU(Xmask[i+1],SaturatedY);
        // Follow a unsaturated path
6.      (F[i+2], XMask[i+3]) =
            MatrixALU(Xmask[i+2], UnsaturatedX);
        // See if any augmenting path is found
7.      PathFound = ReduceOr (XMask[i+3]*
            UnsaturatedYVertices);
8.      i = i+2;
9.  }
```

Figure 10: Detection of augmenting paths algorithm using matrix operations.

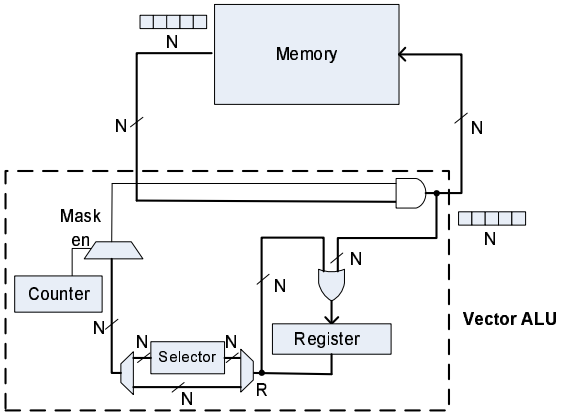


Figure 11: Vector scheduler for the maximum matching algorithm.

```
4.  while Not PathFound and i < K {
        // Follow a saturated path
5.      For(j = 0; j < N; j++) {
6.          (F[i+1][j], XMask[i+2][j]) =
                VectorALU(Xmask[i+1][j],
                SaturatedY[j])
7.      }
        // Follow a unsaturated path
8.      For(j = 0; j < N; j++) {
9.          (F[i+2][j], XMask[i+3][j]) =
                VectorALU(Xmask[i+2][j],
                UnsaturatedX[j])
10.     }
        // See if any augmenting path is found
11.     PathFound = ReduceOr (XMask[i+3] *
            UnsaturatedYVertices )
12.     i = i+2;
13. }
```

Figure 12: Detection of augmenting paths algorithm using vector operations.

YMask is similar to the approach by the Matrix Scheduler; however, it requires N cycles to process a complete matrix and then generate a Mask vector with the Vector Scheduler while it only requires a single cycle with the Matrix Scheduler.

We use VectorALU( ) to represent the function performed by the Vector ALU. The maximum matching algorithm from line 4 to line 12 in Figure 6 is written as shown in Figure 12 by using VectorALU( ).

It requires N cycles to perform one optimization step. However, our performance evaluation results in Section 6 show that this architecture scales the best for large sizes of N.

6 Performance evaluation

We built and evaluated crossbar systems consisting of 16, 32, 64, 128, and 256 nodes in full post synthesis and placement and routing simulation using ModelSim. Using this hardware data extrapolated a functional model for MATLAB simulation of systems of 512 and 1024 nodes for performance estimation. Functional simulation was required in some cases due to limitations of capacity in the full hardware simulation. The hardware cost was evaluated based on Altera Stratix ESP1S25F1020C FPGA chip. The area and performance statistics may vary if different FPGA devices are considered.

If only the scheduling delay is considered, pure combinational logic (Pure Logic Scheduler) and the register array architecture (Matrix Scheduler) have similar performance with

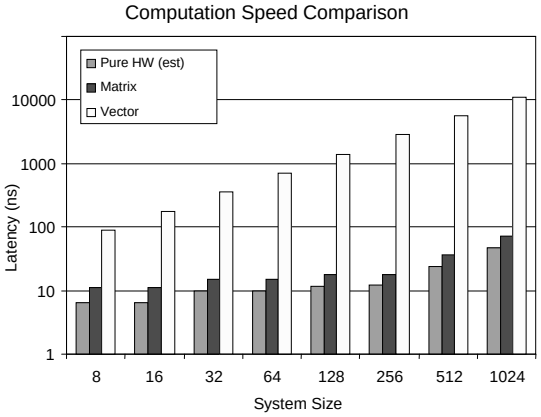


Figure 13: Performance per optimization step. The Pure Hardware performance is based on extrapolated functional simulation. The Vector and Matrix performance numbers are based on actual hardware results ranging from 8-256 and estimated in functional simulation for 512 and 1024.

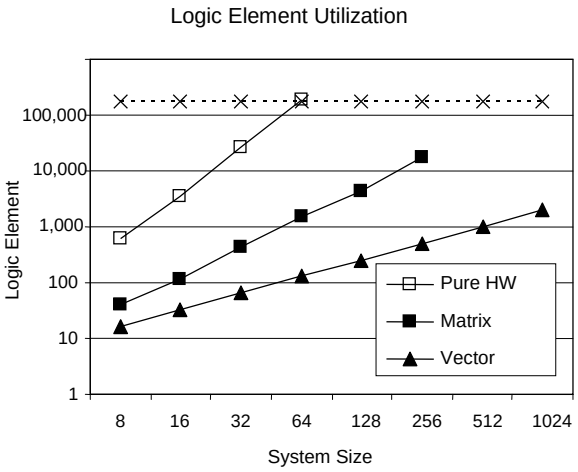


Figure 14: System area cost. The Pure Hardware performance is based on estimations of a single optimization step. The Vector and Matrix performance numbers are based on actual hardware synthesis results ranging from 8-256 and estimated for 512 and 1024.

the matrix version lagging slightly behind in all cases. Both pure logic and matrix schedulers outperform the memory architecture (Vector Scheduler), as shown in Figure 13. However, the area cost of these architectures impacts their feasibility. Figure 14 indicates the area of each architecture in logic elements versus system size, where a dotted line indicates the logic limit of the Altera device. The memory based architecture requires the least logic elements and is potentially scalable beyond 1024 nodes. The register array based architecture consumes somewhat more elements and is limited to a 256 node implementation. The pure logic approach is limited to a maximum of 64 nodes. We examined the memory requirements for adding further optimization steps with the results shown in Figure 15. These results are for the memory based architecture only and again the memory limit of the device is shown with a dashed line. In these results, we see that for K=3 up to K=11, the architecture is implementable for up to 1024 node crossbars. For a fully optimized system where K=2N-1 we are limited to 128 node crossbars.

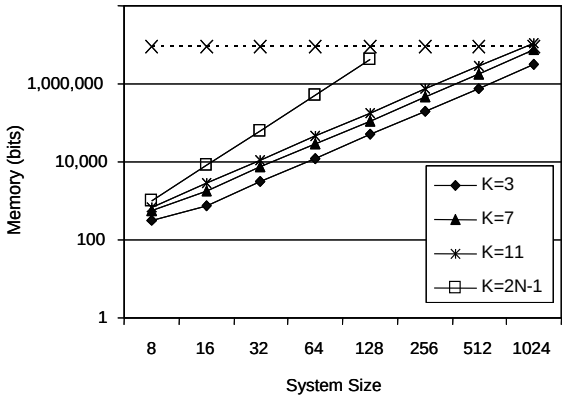
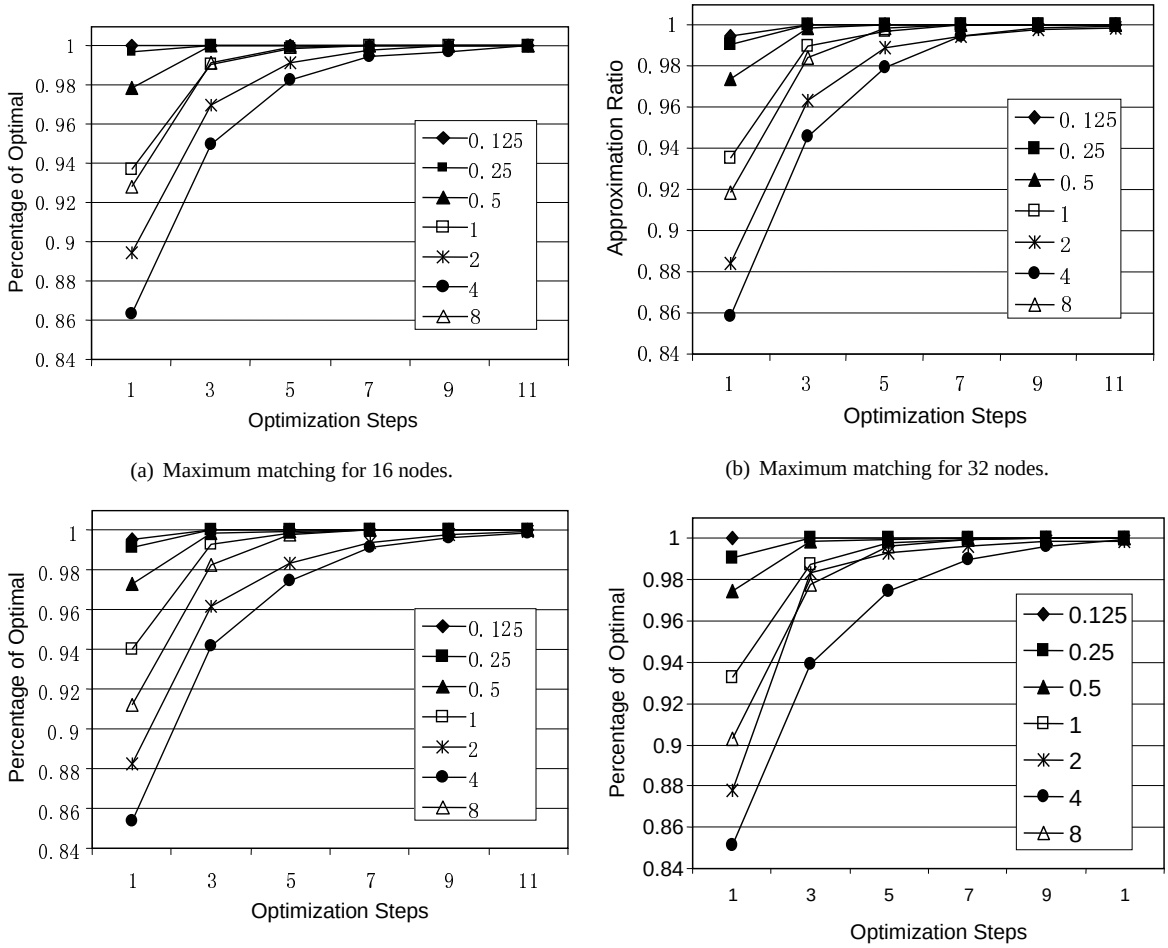


Figure 15: Estimated memory utilization for various step sizes, K, (with K=2N-1 steps being provably optimal.)

We performed system simulations for randomly generated communication matrices with and without complete matching (i.e. communications matrices with and without full permutations of requests) based on system containing 16, 32, 64, and 128 nodes. We generated random communication matrices with various densities. For example, in a 8x8 communication system, if a total of four requests are issued, the density is 4/(8x8), which is 0.125. The experiments are each repeated 100 times and are reported in Figures 16 and 17 as a percentage of optimal. Optimal is calculated by completing the maximum matching algorithm where K=2N-1.

Figure 16 shows the matching performance corresponding to different optimization steps. Each line represents the results generated by communication matrices of a certain density.

The simulation results show that for random requests, the matching degree we get is approximately 99% of optimal matching degree by K=7 for all systems. In order to verify our hardware implementation is capable of finding complete matching, we mixed random communication requests with full permutations to evaluate the system performance. Thus, when a full permutation of requests is included, the optimal result because a fully utilized network. Our proposed algorithm is able to optimize matching successfully. The results show that for mixed requests, our matching degree is approximately 99% of the maximum matching degree after nine optimization steps, as shown in Figure 17.



(a) Maximum matching for 16 nodes.

(b) Maximum matching for 32 nodes.

(c) Maximum matching for 64 nodes.

(d) Maximum matching for 128 nodes.

Figure 16: Maximum matching for randomly generated requests. The different curves represent network load where 0.125 is 12.5% loaded and 8 is 800% overloaded ($K = 1$ represents the initial greedy schedule).

7 Conclusion

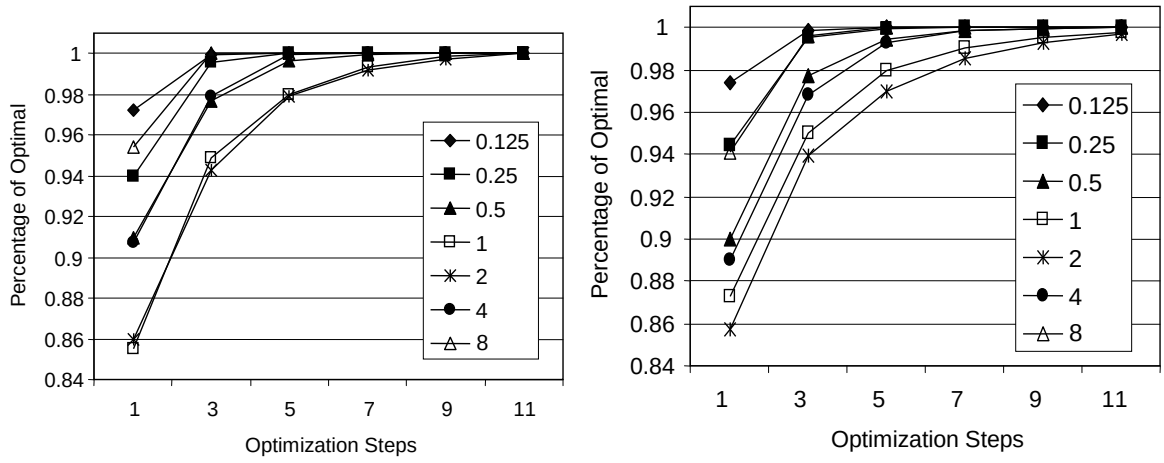
In this paper we present three hardware architectures to solve the maximum matching algorithm in the context of crossbar scheduling in real-time. In our underlying hardware algorithm, we unfold the Bipartite Graph, such that the unsaturated and saturated paths alternate. By using the inherent characteristic of hardware logic, all possible augmenting are detected in parallel with our approach to improve run-time complexity of the system. Based on this fundamental technique we constructed three maximum matching hardware schedulers: Pure Logic, Matrix, and Vector.

As shown in Section 6, these three architectures have an pronounced area performance tradeoff. While, these different architectures are suitable in different circumstances, the matrix scheduler appears to have the best area performance tradeoff, being feasible for up to 256 node crossbars and maintaining ¡ 20ns scheduling delays for this implementation. However, for extremely large crossbars, the vector scheduler is a feasible choice for up to 1024 node crossbars with a scheduling delay of approximately 10 μs, which is a reasonable delay for current electronic circuit switched crossbars. With newer FPGA technologies such as those in 65 nm, the matrix approach could scale up to 1024 node crossbars reducing the scheduling time to under 90 ns.

Some future directions may be to apply these techniques to standard cell or structured ASIC technologies and to examine the power consumption required by these devices. A fast time to market solution would be to implement the vector scheduler with a vector processor and examine the performance trends.

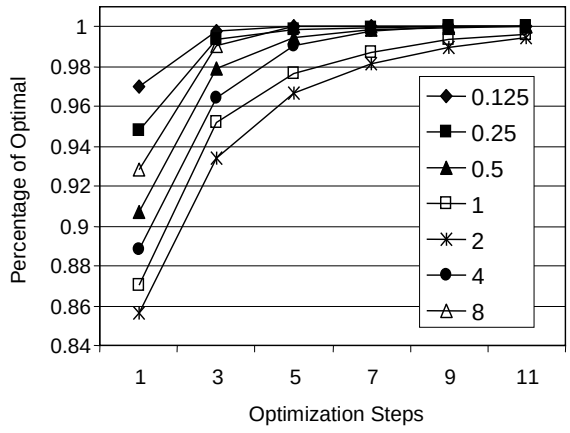
References

F. Glover, “Maximum matching in a convex bipartite graph,” *Naval research Logistics Quarterly*, 1967.

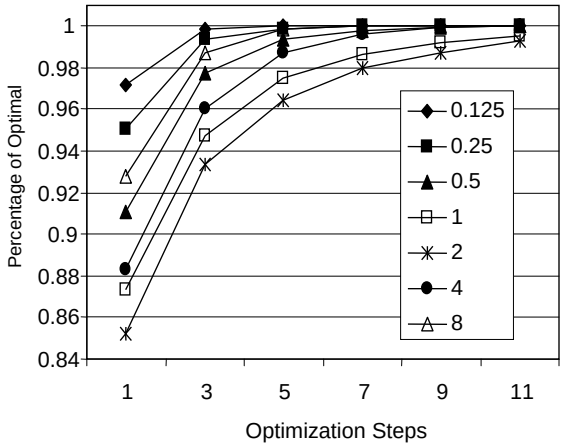


(a) Complete matching for 16 nodes.

(b) Complete matching for 32 nodes.



(c) Complete matching for 64 nodes.



(d) Complete matching for 128 nodes.

Figure 17: Complete matching. Requests are randomly generated requests at variable loads with the oversubscribed network loads being randomly generated permutations ($K = 1$ represents the initial greedy schedule).

A. L. Dulmag and N. S. Mendelsohn, "Actrices associated with the hitchcock problem," *Journal of the ACM*, 1962.

Z. Galil, "Efficient algorithms for finding maximum matching in graphs," *ACM Computing Surveys (CSUR)*, vol. 18, no. 1, pp. 23–38, 1986.

K. Mulmuley, U. V. Vazirani, and V. V. Vazirani, "Matching is as easy as matrix inversion," in *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, (New York, New York, United States), pp. 345–354, 1987.

The BlueGene/L Team, "An overview of the BlueGene/L supercomputer," in *Conf. on High Performance Networking and Computing*, 2002.

S.Habata, M. Yokokawa, and S.Kitawaki, "The Earth simulator system," tech. rep., NEC Res. and Develop. http://www.owlnet.rice.edu/ elec526/handouts / papers/earth-sim-nec.pdf.

Z. Ding, R. Hoare, A. Jones, D. Li, S. Shao, S. Tung, J. Zheng, and R. Melhem, "Switch design to enable predictive multiplexed switching in multiprocessor networks," in *Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, (Denver, CO), Apr. 2005.

N. Mckeown, "Crossbar switch scheduling," *SNRC Talk*, 2001.

T. Weller and B. Hajek, "Scheduling nonuniform traffic in a packet-switching system with small propagation delay," *IEEE/ACM Trans. Netw.*, vol. 5, no. 6, pp. 813–823, 1997.

K. L. Yeung, "Efficent time slot assignment algorithms for TDM hierarchical and nonhierarchical switch systems," *IEEE Trans. on Communications*, vol. 49, pp. 351–359, Feb. 2001.

L.Lovasz and M.Plummer, *Matching Theory*. Academic Press, 1986.

H. N. Gabow, "An efficient implementation of edmonds' algorithm for maximum matching on graphs," *Journal of the ACM (JACM)*, vol. 23, no. 2, pp. 221 – 234, 1976.

A. Czygrinow, M.Hanckowiak, and E.Szymanska, "Distributed algorithm for approximating the maximum matching," *Elservier Science*, 2003.

M. Hanckowiak and M. K. A. Panconesi, "On the distributed complexity of computing maximal matchings," in *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, (San Francisco, California, United States), pp. 219–225, 1998.