Maintaining Bipartite Matchings in the Presence of Failures [†]

Edwin Hsing-Mean Sha
Dept. of Computer Sci. & Engin.
University of Notre Dame
Notre Dame, IN 46556

Kenneth Steiglitz
Dept. of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

We present an on-line distributed reconfiguration algorithm for finding a new maximum matching incrementally after some nodes have failed. Our algorithm is deadlock free, and with k failures maintains at least $M-k$ matching pairs during the reconfiguration process, where M is the size of the original maximum matching. The algorithm tolerates failures that occur during reconfiguration. The worst-case reconfiguration time is $O(k \min(|A|, |B|))$ after k failures, where A and B are the node sets, but simulations show that the average-case reconfiguration time is much better. The algorithm is also simple enough to be implemented in hardware.

[†]This work was supported in part by NSF Grant MIP-8912100, and U. S. Army Research Office-Durham Grant DAAL03-89-K-0074.

1 Introduction

Imagine that there are n persons in Village A and m in Village B. Two persons from different villages can be matched to become a couple, and at any time only one person can be matched to another. Initially the matching is maximum. Sometimes, however, people decide to be alone. Without loss of generality, assume that some in B change their minds. Let $G = (A, B, E)$ be a bipartite graph and $|A| = n$, $|B| = m$. An edge between two nodes means that they are allowed to become a couple. After a person b has changed his or her mind, b's original matching in A must find another available one in B, if possible.

The process of finding a new matching to obtain the maximum number of pairs is called *reconfiguration*. Unfortunately, there is no central agency to perform the reconfiguration process, so this process must be done in a distributed and parallel way. It is also desirable that during the reconfiguration process as many matched pairs be maintained as possible, and that failures during the process be tolerated. Ideally, there should always be at least $M - k$ matching pairs after k persons have changed their minds, where M is the original number of matching pairs. The number of matching pairs should monotonically increase in the reconfiguration process. Therefore, if no new persons change their minds, the reconfiguration process will finally regenerate a new maximum matching, if one is possible.

One motivation for this problem is that such an algorithm can be applied to any fault tolerant system which involves bipartite matching. For example, Kuo and Fuchs [5] show that many problems of spare allocation in VLSI arrays can be modeled as bipartite matching. Based on our bipartite matching algorithm, we can have a distributed reconfiguration mechanism to replace faulty nodes by spare nodes in a redundant array. In [9, 10] highly reliable structures with the asymptotically optimal number of nodes and edges for one-dimensional and tree-like array architectures are given. They use bipartite matchings between levels in layered graphs, and so are particularly well suited for the run-time fault-tolerant algorithm described in this paper.

The general matching problem has been extensively studied. For maximum matching in bipartite graphs, the algorithm of Hopocroft and Karp [3] is the fastest known, and the algorithm by Micali and Vairani [6] is the most efficient one for finding matchings in general graphs. More recently an algorithm for on-line bipartite matching is presented [4]. Papers [8, 11] also give distributed algorithms for maximum matching in general graphs.

Our problem is different from the usual matching problem, which starts with an empty matching. We assume we start with a maximum matching, and after some nodes fail, we would like to have a simple, efficient, and distributed way to find a new maximum matching. Further, the algorithm should start to reconfigure the system as soon as failures occur, even though new failures may occur during the reconfiguration process. We say a reconfiguration algorithm is *on-line* if it can start to reconfigure the system immediately after a failure occurs, and can endure new failures during reconfiguration. This is an especially desirable property for run-time fault tolerance, since the system need not stop to do a reconfiguration process.

We will not be concerned so much with the number of messages that PEs need to send to achieve a new matching, such as is done in the matching algorithms in [8, 11], which in any event are not designed to operate in the presence of faults. Rather, we want to minimize the effects of failures during reconfiguration. Our algorithm does tolerate faults during operation, and ensures that after k failures there are always at least $M - k$ matching pairs, where M is the original number of matching pairs. If there are no further failures, the size of the matching grows monotonically until it becomes maximum. The algorithm is simple enough to be implemented in hardware. The overall reconfiguration time is $O(k \min(|A|, |B|))$ after k failures. The simulation results show that the average-case reconfiguration time is much better.

2 The Basic Ideas of Our Algorithm

We first explain our model. An array architecture is represented by a graph G; each node of G is regarded as a processor, and each edge as a connection between two processors. If nodes have failed, the failed nodes and all the edges incident to them will be removed. If later a failed node is repaired, this node with the corresponding edges will be added to the graph. We assume that if two nodes have not failed, and are connected, they can communicate, that is, we do not model failures of communication.

Definition 2.1 Given a bipartite graph $G = (A, B, E)$, a *matching* M is a subset of the edges such that no two edges in M share the same endnode.

Definition 2.2 If an edge (a, b) is in M, we say that a is b's *matching* node in M, or vice versa. This pair (a, b) is also called matching pair or a matching edge. If no edge in M is connected to node x, we say x is a *free* node.

Definition 2.3 A matching is *maximum* if no other matching of G contains more edges. Given a matching M, an *alternating path* P is a path that does not contain two consecutive edges that are not in M. If an alternating path P starts and ends at free nodes it is an *augmenting path.*

It is well known that M is not a maximum matching if and only if there is an augmenting path. Our algorithm searches for augmenting paths to obtain the maximum matching of G.

After some nodes have failed, the search for augmenting paths to find free nodes will traverse the graph. Basically, our algorithm performs depth-first search for finding free nodes. In this section, we describe our algorithm informally. A formal description of our algorithm is given in the next section. Let $G = (A,\ B,\ E)$ be a bipartite graph. We think of sets A and B as two levels of nodes in a bipartite graph. Initially, we assume that a maximum matching already exists. An initial maximum matching can be obtained from our algorithm in the following way: initially every node in A regards its matching node as failed and starts to run the bipartite matching algorithm. We assume that a failure of a matched node can be detected by its current matching node.

Nodes in both A and B can fail. For failures in B (resp. A), nodes in A (resp. B) will search for free nodes. We have two versions of our algorithm: Version A is for failures in B and Version B is for failures in A. These versions are the same except A and B are interchanged. However, if our algorithm is to be used as a reconfiguration algorithm for the layered fault-tolerant structure in [10], we only need the Version A because each layer can be regarded as level A.

Let a be a matched node in A and b be a's matching node in B. If node b fails, Version A of our reconfiguration starts at node a. Node a becomes what we call a *super*

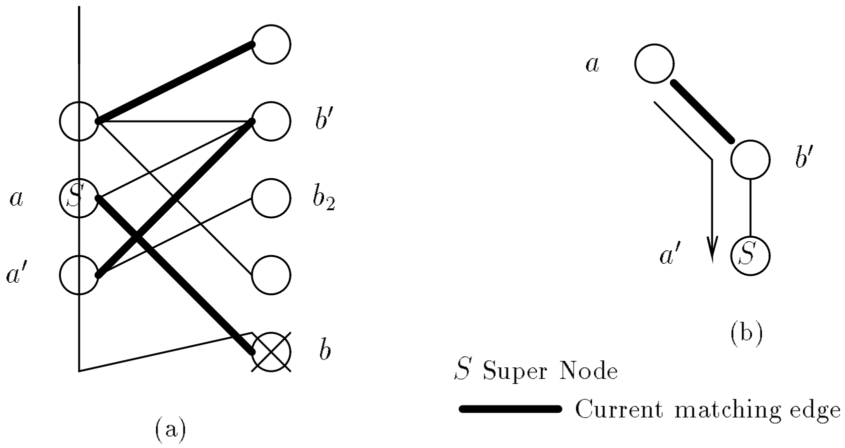

Figure 1: The figure for passing super nodes.

node because it has the privilege of choosing a good node to be its matching node. If a node in A fails, the matching node of this failed node will become a super node to initiate Version B of our reconfiguration algorithm. These two versions of our algorithm are performed independently to obtain a maximum matching. In this section, without loss of generality, we only explain Version A. However, we need to show that the failures in A do not affect the correctness of the Version A. Here we explain what the actions a super node a will do.

First, super node a tries to find a free node in B which is connected to a. If this node is available, it becomes a's matching node. Otherwise, super node a will try to steal a node which is already matched to another node in A. For example in figure 1, after node b fails, a becomes a super node. Since there is no free node connected to a, a will steal node b' which was matched to a'.

Definition 2.4 If a super node x chooses a node y which has been matched to x' to be its new matching node, we say that x steals y from x'.

After b' has been *stolen* by a, node a' will become a super node because a' does not have a matching node. We can think of this process as the token of *super node* traversing the path from node a to node a' (Figure 1 (b)). A *root node* is a node that initiates a search process for finding a new matching after its matching node becomes faulty. The root node is the first super node in a search process. There may be several searches going on simultaneously, each having a root node.

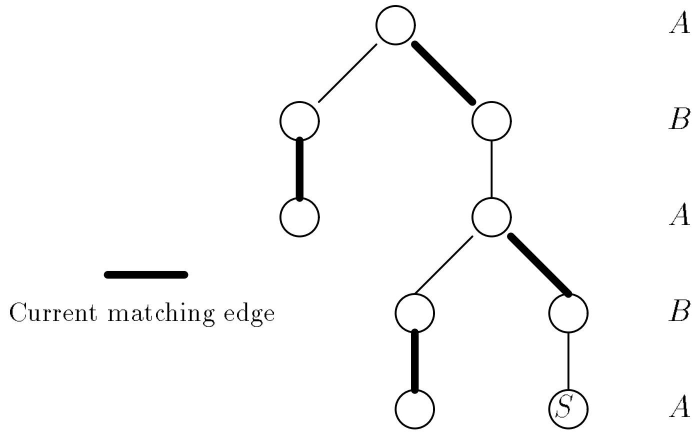

Figure 2: An example of alternating tree.

Our algorithm does a depth-first search (DFS) for finding augmenting paths[7]. The process of searching can be represented as a search tree called an *alternating tree*. A typical alternating tree is shown in figure 2. Each root node is the root of an alternating tree, and at any time a super node is associated with the node which is performing DFS in a tree. There will be precisely one super node in each alternating tree. A new matching is found when a super node acquires a free node. To prevent cycles in searching, we can simply store a bit in each node b to indicate if it has been *reached*. We say that this node is marked *reached*. When a super node finds a free node, this super node sends messages to unmark the corresponding nodes, as explained later in this section.

If a super node at a particular point cannot find an adjacent free node, and finds that all the adjacent nodes are marked *reached* (either by this tree search or some other), it backtracks immediately. Under backtracking, some super nodes may backtrack to root nodes, and these super nodes remain there in an idle state. Thus, we need a way to reactivate when some other super nodes find free nodes. After a super node has found a free node, this super node sends a message, called *UNMARK_BACKTRACK*, recursively to unmark all the nodes which have been passed through by a backtracking super node along an alternating path. For example, in figure 4 there are two idle super nodes, $S1$ and $S2$. After $S3$ has found a free node, $S3$ will send the message, *UNMARK_BACKTRACK* to wake up the idle super nodes $S1$ and $S2$.

Version A and Version B of our algorithms are performed alternatively. In each version, there are three phases as shown in figure 3. Every node performs the same

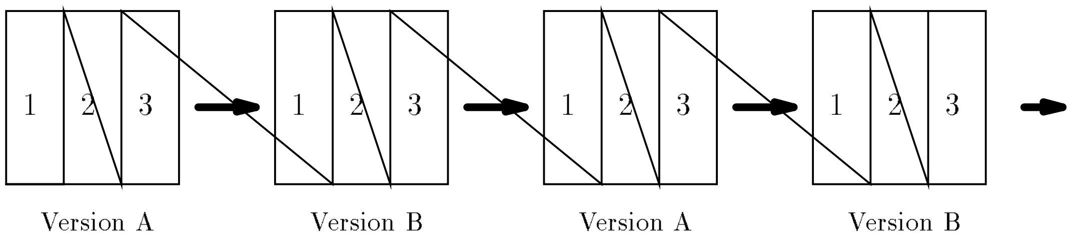

Figure 3: A running sequence of our algorithm; each version has three phases

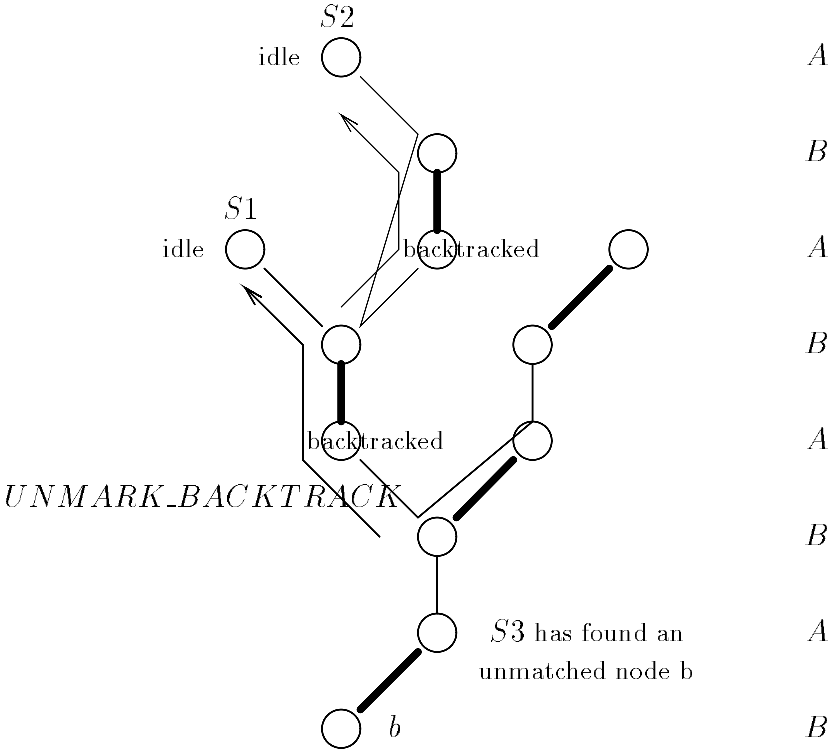

Figure 4: An example of breaking idleness

phase in the same version. Therefore, we need to synchronize all the nodes to perform the same version and the same phase. One possible implementation is to use common wires connected to every node. Because we consider our algorithm to be performed in tightly-coupled processor arrays, few wires connected to every node (PE) are practical assumptions. We can assume there are three signal wires connected to every node (PE). Wire w_{CLOCK} is the clock wire to synchronize the phases of a clock. Wires w_A and w_B are to indicate which version is running. When w_A (resp. w_B) is high, Version A (resp. B) is running. If we do not want to use these common wires, we can use more complicated message passing protocol for synchronization [1].

3 Our Reconfiguration Algorithm

In this section, we explain our algorithm. Since Version A and Version B are essentially the same, we only present Version A in this section. First we define some terms for Version A of our algorithm.

Definition 3.1 The node *old(n)* is n's original matching node before the reconfiguration, and the node *cur(n)* is n's current matching node during reconfiguration.

Initially, for every node n, we set $cur(n) = old(n)$.

In our algorithm, there are several attributes for nodes in A and B, which are used and set during the operation of the algorithm. First, any node is *good* if it has not malfunctioned. The attributes of a node b in B are summarized as follows. A node $b \in B$ is

- *free* if it has no matching node under the current matching,
- *reached* if it has been reached by some depth-first-search in our algorithm. When a node is not reached, we say that this node is *unreached.*

The attributes of a node $a \in A$ which is reached by some search process can be marked by message passing as follows. Node $a \in A$ is

- *super* if $cur(a)$ is not good, or it is unmatched because its matching node $cur(a)$ has been stolen by some other node,
- *backtracked* if a search that reaches node a finishes searching node a's subtree and must backtrack to a's parent.

We call a node super if and only if it has a *supernode token.* This token can be transferred to other nodes along the depth-first-search traversed in our algorithm. Messages need to be passed in our algorithm for changing the current states of nodes $a \in A$. There are three messages that can be sent: *SUPERNODE*, *UNMARK_BACKTRACK*, and *CHANGE_OLD_MATCHING*. We discuss these three messages one by one as follows.

(1) The message *SUPERNODE* represents the super node token. If node a receives the message *SUPERNODE*, a becomes the super node. There are two situations when a node a sends this message. The first situation is when node a steals some other's matching node. The second situation will be explained later in the section (see figure 5).

(2) After a super node s has found a free and good node in B, s will send the message *UNMARK_BACKTRACK* to all the backtracked nodes which are adjacent to node $old(s)$. This message is used to set some nodes in B as not *reached* so that some idle super nodes can start to search for free nodes. When a node $a \in A$ receives the message UNMARK_BACKTRACK, a will set node $old(a)$ as *unreached*. Then, after a sends UNMARK_BACKTRACK to $old(a)$, $old(a)$ will immediately send this message to all the backtracked nodes which are adjacent to $old(a)$.

(3) When a super node finds a free and good node in B, this super node will send the message *CHANGE_OLD_MATCHING* to the nodes in the alternating path so that their old matching nodes are set to be the current matching nodes. When a node a gets the message CHANGE_OLD_MATCHING, node a will mark the node $old(a)$ as unreached, and ask $old(a)$ to send UNMARK_BACKTRACK to all the backtracked nodes which are adjacent to $old(a)$.

Our algorithm runs in parallel at all the nodes. Initially, there is a bipartite maximum matching. In Phase 1, each node checks if it needs to initiate a searching process because of the failure of its current matching node.

The real search process is performed in Phase 2. If the super node a is successful in finding a free and good node, a sends messages CHANGE_OLD_MATCHING and UNMARK_BACKTRACK as we explained previously. If node a cannot find a free node, node a will try to steal others' matching nodes. The super node a will steal an unreached and good node b, and send the message SUPERNODE to node $cur(b)$. Otherwise, if all a's adjacent nodes have been marked *reached* and the node $old(a)$ is good, a will backtrack. Node a will retain its old matching node and send SUPERNODE to node $cur(old(a))$. Otherwise, if the supernode token has backtracked to a root node, this supernode token will wait there.

In Phase 3, node a will do the appropriate operations depending on which message a has received. If there are failures in A, their corresponding old matching nodes become super nodes. We will explain the details later. In Version A, a super node in A should not steal any super node in B, since these super nodes in B will start their searches later in Version B. Denote by N the node which is performing the following algorithm. The following is a sketch of Version A of our algorithm which runs at all the nodes in A in parallel. A more detailed algorithm is presented in the appendix.

```
/* Let set E be the set of nodes in B which are good, adjacent to N,
and not super nodes.  */
  Phase 1
      If cur(N) is not good, N is a super node.

  Phase 2
  If N is a super node
      If there exists a free node in E
          Set old(N) to be not reached
          Ask old(N) to send CHANGE_OLD_MATCHING to cur(old(N))
          Ask old(N) to send UNMARK_BACKTRACK to all adjacent backtracked nodes
      Else if N can steal an unreached node b in E
          Ask b to send SUPERNODE to cur(b)
      Else if old(N) is good
          backtrack from N
      Else
          Do nothing

  Phase 3
  If N receives SUPERNODE
      Set N to be a super node.
  If N receives CHANGE_OLD_MATCHING
      Set old(N) to be not reached
      Ask old(N) to send CHANGE_OLD_MATCHING to cur(old(N))
      Ask old(N) to send UNMARK_BACKTRACK to all adjacent backtracked nodes
  If N receives UNMARK_BACKTRACK
      Set old(N) to be not reached
      Ask old(N) to send UNMARK_BACKTRACK to all adjacent backtracked nodes
```

We would like to discuss the operations which nodes in B perform in Version A. We need define the following terms.

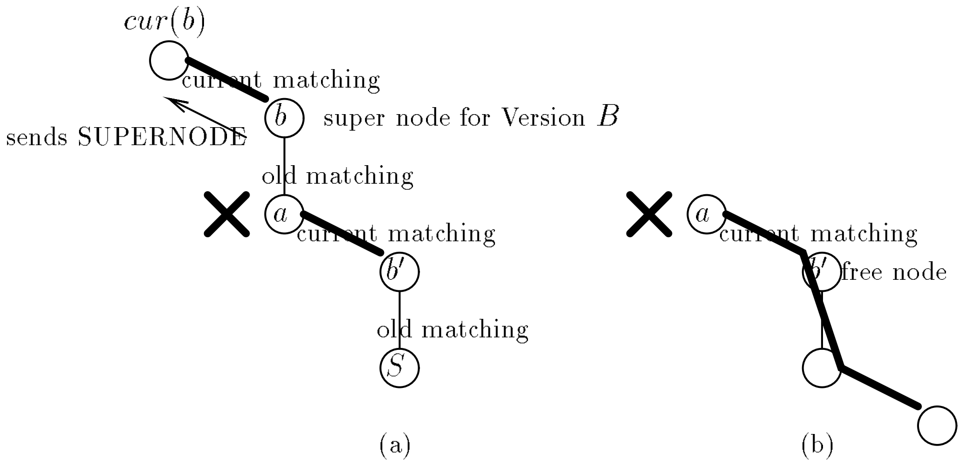

Figure 5: A failure in A which is in an active alternating path

Definition 3.2 A super node a is called *idle* if node a is a super node and every adjacent node of a is labeled *reached*, and $old(a)$ is not good; otherwise a super node is called *active*. We say an alternating path is *active* if the corresponding super node is active.

In Version A, nodes in B basically perform the message passing for nodes in A. However, when there are failures in A, the old matching nodes of these failures become super nodes. These super nodes in B do not perform any search while the algorithm is running Version A, but they need to do some operations for nodes in A.

There are two cases for failure of a node a in A: either a is not in an active alternating path or a is. Let b be the old matching node of a. If a is not in an active alternating path, b will do nothing except become a super node for Version B. If a is in an active alternating path as figure 5 (a) shows, b becomes a super node and initiates a backtracking to $cur(b)$ (b sends SUPERNODE to $cur(b)$). This backtracking is to restore the alternating path. We can regard the original search as not passing through b because b has now become a super node for version B.

There is a detail here. Let b' be the current matching node of a. In the proof of Lemma 4.2, we will show that failures of nodes in A do not affect the correctness of Version A. To ensure this, node b' should be set free if the super node of the alternating path, say S, finally finds a free node. Figure 5 (b) shows this case. Figure 7 shows another case when S cannot find a free node.

For nodes b in B:

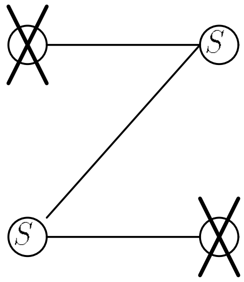

Figure 6: The case that two super nodes acquire each other

```
If old(b) is not good
    Set b to be the super node.
    If old(b) is not the same as cur(b)
        Send SUPERNODE to cur(b).

If b receives CHANGE_OLD_MATCHING and cur(b) is not good
    Set b to be free.
```

Before we present the overall algorithm, we need a definition.

Definition 3.3 We say that A in G (resp. B in G) is *stable* if there is no active super node in A (resp. B). If both A and B in G are stable, we say the graph G is stable.

When the graph is stable, there are no active super nodes, so the reconfiguration algorithm is over. In Version A, super nodes in B, because of the failures in A, will not be acquired by any super node in A in Phase 2. Therefore, we need to consider the case when two super nodes should acquire each other to increase the matching size as figure 6 shows. After A is stable, we will set all the idle super nodes in A to be free so that the super nodes in B can acquire them in Version B.

The overall algorithm

```
repeat
    repeat Version A until A is stable.
    All the idle super nodes in A are set to be free.
    All the nodes in B are set to be unreached.
    repeat Version B until B is stable.
    All the idle super nodes in B are set to be free.
    All the nodes in A are set to be unreached.
until graph G is stable.
```

4 The Analysis and Proof of Correctness

We prove the following lemmas to obtain more insight into this algorithm. In a stable graph, there are no active super nodes. Thus, when a graph is stable, our reconfiguration algorithm is over. We would like to prove that if a stable graph is reached by using our algorithm then a maximum matching has been found.

When there is only one super node, our algorithm is the same as the standard sequential algorithm which performs depth-first search to construct the alternating tree. Therefore, the following is easily proved.

Lemma 4.1 When the number of active super nodes is one, this algorithm correctly finds a maximum bipartite matching. $\Box$

When there is more than one super node, the situation becomes more complicated. After some nodes have failed, during reconfiguration some current matching nodes of good nodes may not be the same as their old matching nodes. If the current matching node of node a' fails later, node a' will initiate a new search process (reconfiguration process). Thus, we need to consider the situation where new failures occur during a reconfiguration process. The next lemma shows that it is sufficient to consider the case where all failures that occur in running a version of our algorithm occur at the very beginning of running that version.

Lemma 4.2 If Version A of our algorithm works when all nodes fail at the beginning, then Version A of our algorithm works when some nodes fail during the reconfiguration.

Proof: We proceed by induction of the total number of failures during a particular invocation of Version A. Suppose that there is already a reconfiguration process for k failures and then suppose that a new failure occurs. We would like to show that the above situation can be regarded as these $k+1$ nodes failing at the beginning. We first consider the failures in A. If a node in A, say a, fails, there are two cases: a is not in an active alternating path or a is.

Case A1: We first consider the case when a is not in an active alternating path. Let b be the old matching node of a. then b will become a super node for Version B. Since b's old matching node is the same as its current matching node, the message of

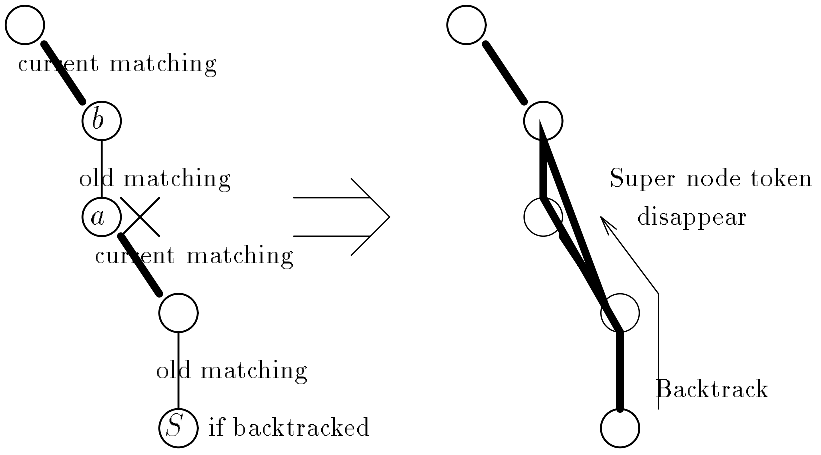

Figure 7: A failure in nodes A

SUPERNODE will not be sent for this failure. It is obvious that this failure can be regarded as if it had occurred at the beginning.

Case A2: If a fails in an active alternating path. the message SUPERNODE will be sent from b to $cur(b)$. We can regard the original alternating path as searching to $cur(b)$ and cannot go through b because b was a super node at the beginning. If a is not a super node, in this case it looks like the number of super nodes becomes one more than the number failures. However, we will show that the super node in the alternating path starting from a is redundant and will disappear later. The super node in the alternating path starting from a either finds a free node or not. If it does not find a free node, after the supernode token is backtracked to the failed a, the supernode token will disappear as shown in figure 7 (b). If it finds a free node, b' will be set free later as shown in figure 5 (b). Whether it finds a free node or not, the size of the current matching does not change in the end. Thus, this super node is redundant and will disappear.

The preceding argument suffices when a is not a super node. We now need to consider the cases when a is a super node, and it fails after some messages have been sent. If a fails after having sent SUPERNODE, it is similar to Case A1. If the message CHANGE_OLD_MATCHING has been sent before a fails, this case is similar to Case A2. Therefore, we have shown all the cases for failure of node a.

Now we consider failures in B. First, we describe the general idea. We analyze the cases where failures happen at different instructions in our algorithm. The details for

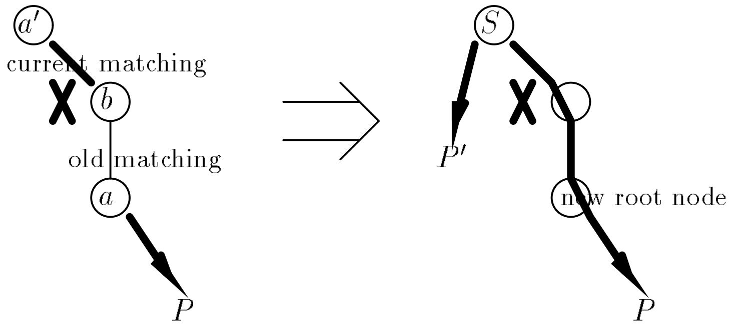

Figure 8: The figure for case 2

different cases are given in Appendix B. Let b be the new faulty node, a be the old matching node of b, and a' be the current matching node of b. There are two cases for the new faulty node.

Case B1: If a is the same as a', node a backtracks or has not been reached. Since $old(a)$ is not changed by the reconfiguration process, this failure can be regarded as if it had occurred at the beginning with the other k failures.

Case B2: If a is not the same as a', a and a' must both be in an active alternating path, say P, and a's old matching node b must have been stolen by a' as shown in figure 8. From our algorithm, we know that a' will become a new super node and will start a new search P' because of the failure of b. Node a becomes a root node for the search P, because a's old matching node b has failed, and a cannot backtrack farther back to a'. The new search P' starting from a' can be regarded as the original search P reaching a'. Thus, the new failure can be regarded as if it had occurred at the beginning with the other k failures.

As mentioned the details of the analysis for different situations are shown in the appendix. After the above observations, the lemma is proved by induction on k. $\square$

Thus, without loss of generality, we can assume that all the failures in running a version of our algorithm happen at the same time. We know that if a super node has no way to proceed with the search, it backtracks. When a super node backtracks to a root node (their old matching nodes are not good) and every adjacent node has been marked reached, this super node is idle. Therefore, the following observation follows easily from our algorithm.

Lemma 4.3 An idle super node must be a root node. □

The next lemma shows that if we cannot find an augmenting path from any root node, then a maximum matching has been obtained.

Lemma 4.4 After a graph is stable, there is no augmenting path from any root node if and only if a maximum matching has been obtained.

Proof: The if-part is obviously correct. We prove the only-if-part. Initially, a maximum matching is assumed. If we remove the failed nodes and their corresponding matching nodes, it is obvious that the matching of the remaining nodes is still maximum. We know that every failure at a matched node creates a root node either in A or in B. Therefore, if there is an augmenting path to increase the size of matching, we can always find an augmenting path starting from a root node. In other words, a maximum matching is obtained if there is no augmenting path starting from any root node. □

Let G be the graph after we delete the failed nodes and their incident edges. Let S_B be the set of super nodes in B, and $G - S_B$ be the remaining graph after the nodes in S_B and their incident edges are deleted from G. The graph $G - S_A$ is similarly defined. The next lemma shows the property of the current matching after the first Version A is finished.

Lemma 4.5 After the first Version A is finished, the current matching is maximum for $G - S_B$.

Proof: From lemma 4.4, we know that if there is no augmenting path starting from any root node, then a maximum matching has been obtained. The first Version A is finished only when A is stable. In the graph $G - S_B$, there is no super node in B. Thus, if A is stable, $G - S_B$ must be stable. Therefore, we can prove the lemma by showing that when A is stable, there does not exist an augmenting path starting from any remaining root node in A. Thus, a maximum matching is obtained for $G - S_B$.

We consider the graph $G - S_B$. From lemma 4.2, we know that we can assume that all failures of nodes occur at the same time. We prove this lemma by induction on the number of root nodes in A, say k. When $k = 1$, it is true by lemma 4.1. Assume that the lemma is correct when the number of root nodes is less than k. Let M_k be the matching

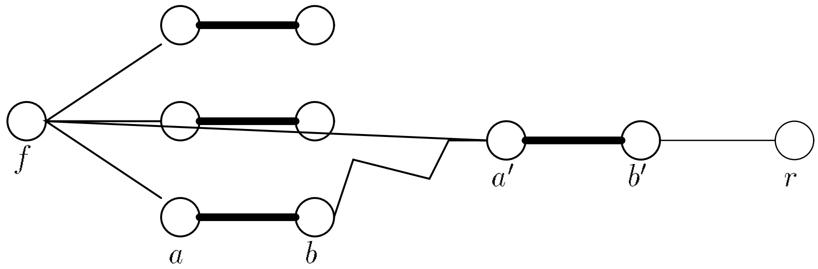

Figure 9: Alternating path for showing impossibility of all super nodes being idle

found in Version A. If M_k is maximum, the theorem is proved. We claim that if M_k is not maximum then there must exist at least one root node which will find a free node. We will prove this claim later. From the induction hypothesis and this claim, we know that when $G - S_B$ is stable, there does not exist any augmenting path starting from a root node in A, and the lemma is proved.

Now we prove our claim. Assume that all those k root nodes are in one connected graph. Otherwise, there exists a smaller subgraph, so we can use the induction hypothesis to prove the theorem. Suppose that Version A is stable, i.e. every super node in A is idle. From lemma 4.3, we know that every super node goes to a root node. Assume that f is a free node in B which would have been in a new matching. Consider an alternating tree rooted at f. There must exist one alternating path from f to a root node r. Under this alternating path, say that a is the node adjacent to f, and b is a's old matching node (see figure 9). Since f is not found by a super node, b must not be reached by any super node. With the same argument, all the nodes in B appearing in the alternating path cannot be reached. This argument shows that there must exist one node adjacent to r which is unreached. Thus, the super node should not be idle, and Version A would not be stable, either. Therefore, at least one root node would find a free node. Our claim is proved.

Since the super nodes will not all be idle, at least one will find a free node, and then this event will unmark the corresponding backtracked nodes to wake up some idle super nodes, if they exist. By repeating the above argument, the number of active super nodes decreases monotonically until there is no augmenting path starting from any root node in A. $\square$

Now we prove the main theorem which shows the correctness of our algorithm.

Theorem 4.6 By using our reconfiguration algorithm if a stable graph is reached then a maximum bipartite matching is obtained.

Proof: Our algorithm alternatively performs Version A and Version B until the graph is stable. Let G be the graph obtained by deleting the failed nodes and their incident edges. Note that if a new failure occurs, G will be changed accordingly. From lemma 4.5, we know that after the first Version A is performed, we have a maximum matching for graph $G - S_B$. If we delete all the failed nodes and nodes in S_B, the matching after the first Version A is finished will be a maximum matching. Thus, if we can increase the matching size, we must be able to find an alternating path starting from a node in S_B. Therefore, in a way similar to the proof in lemma 4.5, we can prove that after Version B (resp. A) is performed, a maximum matching for the graph $G - S_A$ (resp. $G - S_B$) is obtained.

In our algorithm after Version A (resp. B) is finished, all the idle super nodes in A (resp. B) are set to be free. Without loss of generality, we assume that the last version which is performed is Version B. It is obvious that no super node in A is created in running the last Version B. Thus, S_A is empty. Because S_A is empty, B is stable if and only if G is stable. We know that if B is stable, then a maximum matching for $G - S_A = G$ is obtained. Therefore, a maximum matching is obtained for G if a stable graph is reached. $\square$

We assume that the actions in a phase can be finished in unit clock time where a *clock* consists of three phases. Note that for a more detailed timing analysis, the parallel running time for each phase is actually upper bounded by $O(d)$, where d is the maximum degree of a node.

Theorem 4.7 The total number of clock ticks for a reconfiguration to find a bipartite matching for k failures in a bipartite graph $G = (A, B, E)$ is $O(k \min(|A|, |B|))$ by our algorithm.

Proof: We know that all k super nodes will not be idle at one time. The worst case happens when $k - 1$ super nodes are idle. and only one super node is active. All the nodes in A or B can be searched at most twice, once forward and once backtracked. Thus, in $O(\min(|A|, |B|))$ clock ticks, one super node should find a free node. After the completion of the first root node, it unmarks the corresponding backtracked nodes which takes $O(\min(|A|, |B|))$ clock ticks. The worst case occurs when only one idle super node wakes up to perform a search again. Thus, the total maximum number of clock ticks is $O(k \min(|A|, |B|))$. $\square$

In [10], the fault tolerant graph is constructed by using N levels, and every two levels are connected by a bipartite matching. Suppose that there are N levels and each level has $m = O(\log N)$, nodes in our d-degree expander architecture. We analyze the worst case performance, although it is very unlikely to happen. The worst case happens when the faults are in the first level, since we need to propagate the new matchings through the N levels. The total reconfiguration time is $O(kmN)$ if k failures occur. Suppose M packets (or operations) need to be sent in one pipeline. Usually, M is very large compared to N. The minimum time to finish the whole operations is $M + N$. Suppose k faulty nodes occur in the operation. According to the above theorem, the total operation time is at most $M + N + O(kmN)$.

5 Expected Reconfiguration Time

In this section we would like to show that the expected number of clock ticks to finish a reconfiguration is small. We assume that the size of a maximum matching is $\alpha|B|$, where $\alpha < 1$. Considering random graphs with maximum degree d, when there is one failure, reconfiguration time is actually upper bounded by a constant value.

Lemma 5.1 The expected reconfiguration time of our algorithm for one failure is at most $\dfrac{2}{1-\alpha^d}$,

Proof: Let M be the set of matched nodes in B, and Y be the set of free nodes in A. Since $|M| = \alpha|B|$, the probability that a node in A is only connected to M is α^d. In each clock, if a super node s cannot find a free node, it either steals another matched node or backtracks. Let *search tree* S be a tree by merging nodes in B and their corresponding matching nodes in an alternating tree, and the size of search tree be the number of nodes in S. When s steals another matched node, the size of a search tree grows one. When s backtracks, the size of a search tree remains the same. However, all the edges in search tree can only be traversed at most twice. Thus, we know that the number of clock ticks for reconfiguration is at most two $\times$ the size of the search tree.

If the size of a search tree is k, we know that $k-1$ nodes are only connected to M, and there exists at least an edge from the remaining one node to some node in $B-M$. The probability that the size of a search tree is k is $\alpha^{d(k-1)}(1-\alpha^d)$. Thus the expected size of a search tree is

$$\sum_{k=1}^{\infty} k\alpha^{d(k-1)}(1-\alpha^d) = \frac{1}{1-\alpha^d}.$$

Therefore, the expected reconfiguration time of our algorithm for one failure is at most $\frac{2}{1-\alpha^d}$. $\square$

For example, when $\alpha = 8/13$ as our first simulation assumes in the next section, the expected number of clock ticks of reconfiguration for one failure is at most 2.2 if the maximum degree d is 5.

6 Experimental Results

In this section, we show some experimental results for multiple faults. We first explain the first two simulations. The results in the first two simulations show that the average reconfiguration time is within several clock ticks. We construct random bipartite graphs and simulate the algorithm ten thousand times for different numbers of failures. Failures are randomly produced in the beginning of a simulation. In the first simulation, there are 40 nodes in A and 65 nodes in B. Three sets of results are given by different maximum degrees of nodes, 4 (solid line), 5 (dashed line) and 6 (dotted line). We show the average number of clock ticks which are needed to completely finish reconfiguration in Figure 10. Because a graph with higher degree has more possibility to find free nodes than a graph with lower degree, the average reconfiguration time for a graph with higher degree is less than the time for a graph with lower degree.

In the second simulation, there are 400 nodes in A and 450 nodes in B. Three results are given in Figure 11 for different maximum degrees of nodes — 10 (solid line), 14 (dashed line) and 18 (dotted line). From these simulations, we know that only several clock ticks are needed in average to complete a reconfiguration.

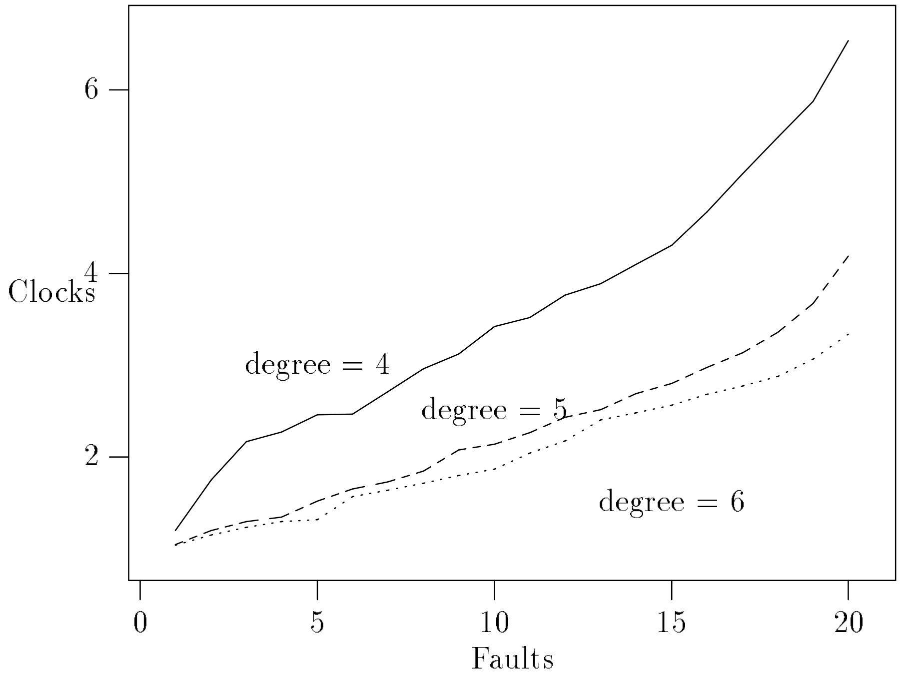

Figure 10: The results of Simulation 1

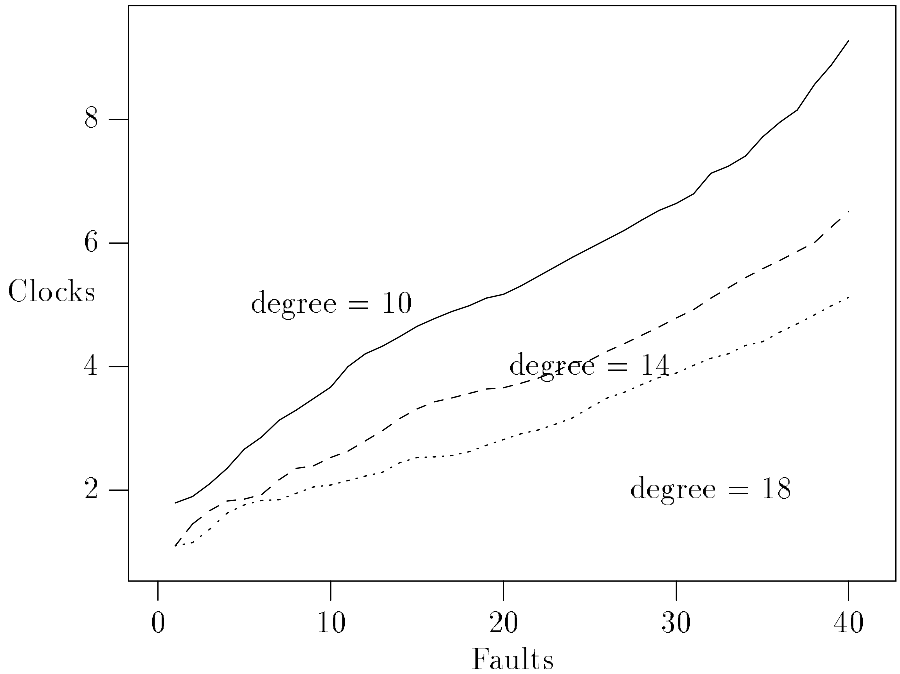

Figure 11: The results of Simulation 2

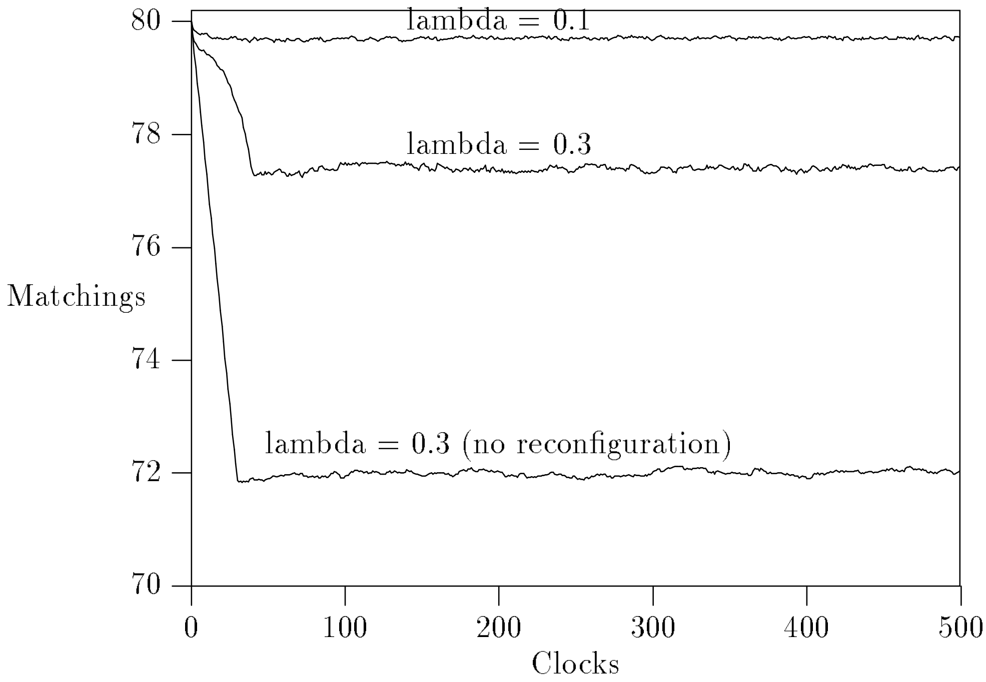

Figure 12: The simulation of repairable failures with different lambdas

We also simulate the situations where failures occur during reconfigurations. The occurrence of a failure follows a Poisson process at each clock. Let *lambda* be the probability that a failure occurs in a clock. In the following simulations, there are 80 nodes in A and 90 nodes in B in this simulation, and initially the size of the maximum matching is 80. We simulate the situations that failed nodes can be repaired. Let the *repair period* be the number of clock ticks that is sufficient to repair a failed node. Thus, if a node b is failed at clock t, at clock ($t+$ repair period) node b is repaired. In this simulation, we set repair period to be 30. Figure 12 shows the average matching size during each clock. When lambda is 0.1, the average matching size is always almost 80, and when lambda is 0.3, the average *stable* matching size is about 77.5 after 50 clocks. If we don't use our reconfiguration algorithm, the average stable matching size is only about 72.

The simulation in Figure 13 shows the average matching size with two different repair periods, 30 and 60 with given a lambda = 0.3. Figure 13 shows when repair period is 60, the average stable matching size is dropped to about 71, but still much better than the stable size without reconfiguration.

The following two simulations show the average stable matching sizes for different

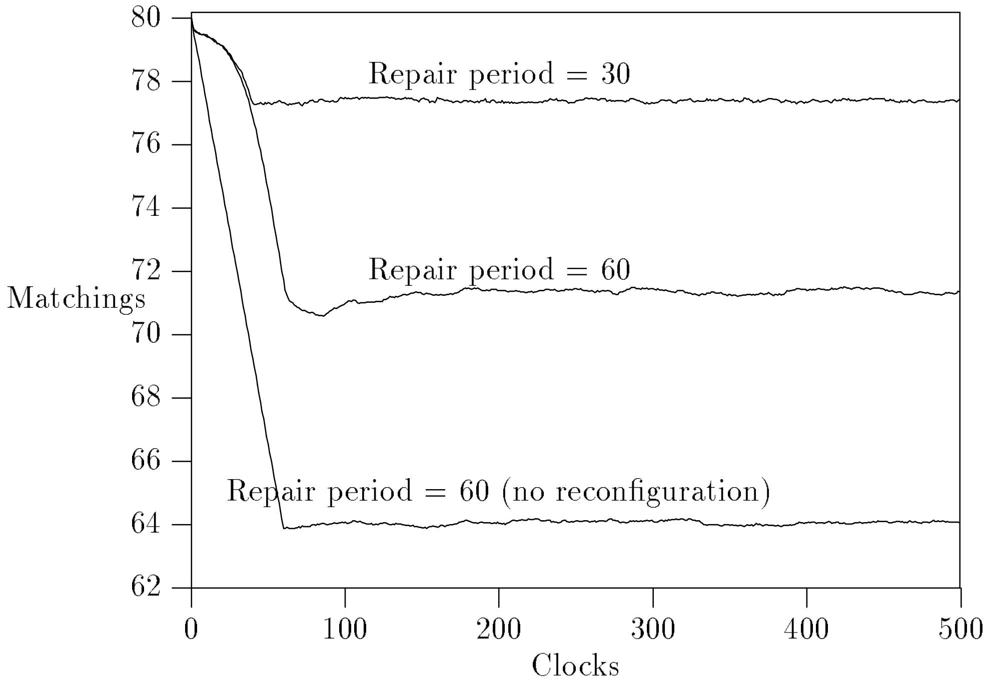

Figure 13: The simulation of repairable failures with different repair periods

lambdas and repair periods. We set the repair period to be 30. Figure 14 shows that the average stable matching size is larger while lambda is smaller. Therefore, when lambda is not too large, say less than 0.2, which is a practical assumption, the matching size is about 79. It also shows that by doing our reconfiguration algorithm, the matching size is increased significantly.

We set lambda to be 0.1 in the last simulation. Figure 15 shows that when repair period becomes longer, the average stable matching size is smaller. However, the stable matching size is always much better than the size with no reconfiguration.

7 Conclusion

Given a bipartite graph $G = (A, B, E)$, nodes in A and B are regarded as processor elements which can fail at any time. After nodes have failed, the failed nodes and their adjacent edges are deleted. This paper presented an on-line distributed algorithm to find a maximum bipartite matching after some nodes have failed. This algorithm can tolerate failures during reconfiguration. Since this distributed algorithm finds a new matching incrementally, it is especially suitable for reconfiguration in run-time fault tolerance.

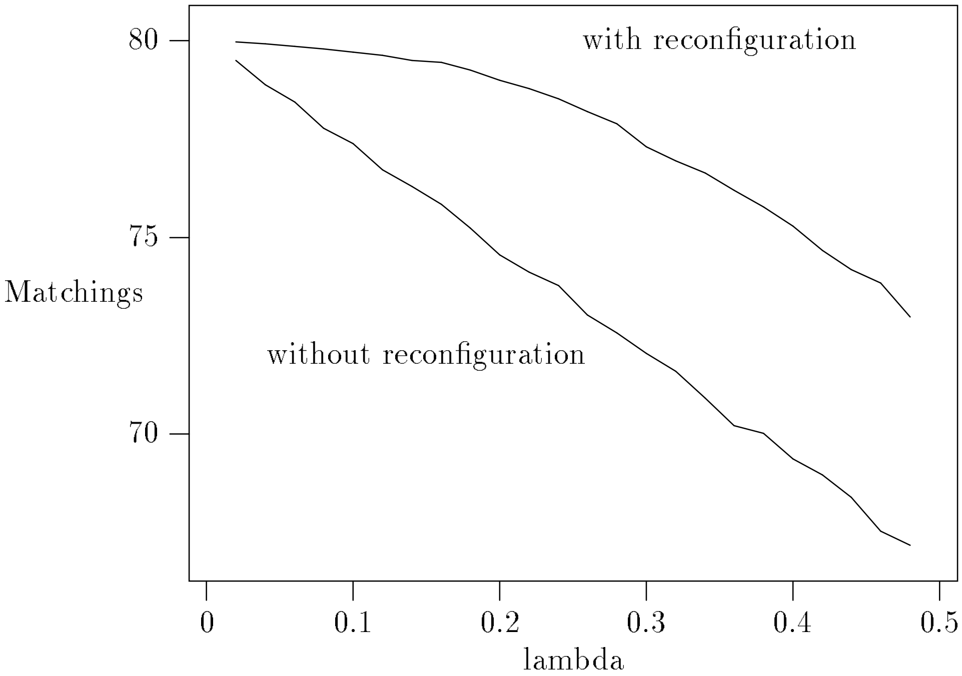

Figure 14: The matching size with different lambdas

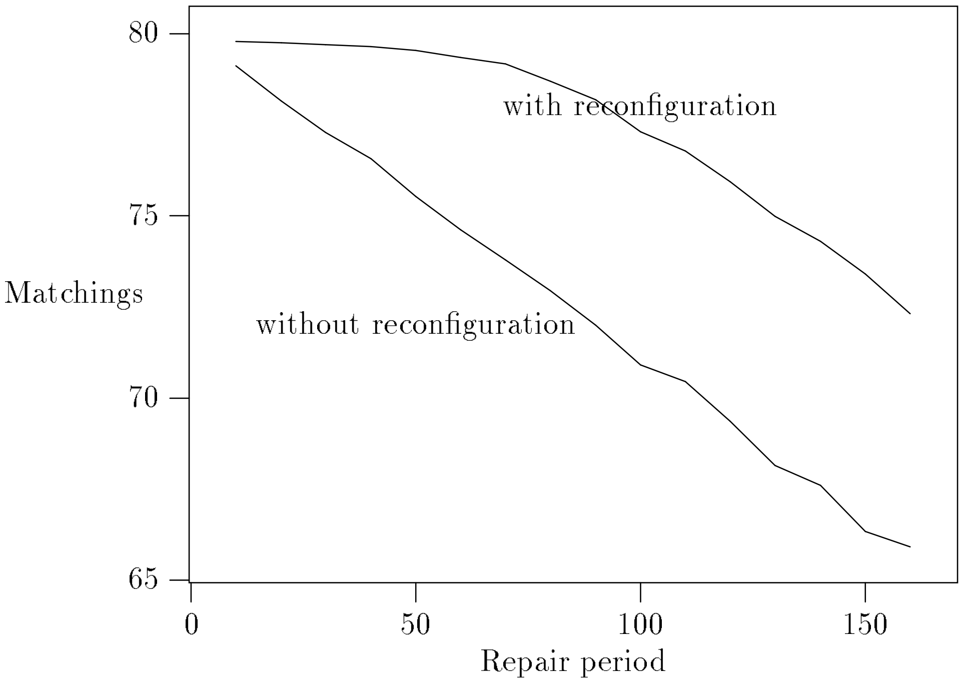

Figure 15: The matching size with different repair periods

The size of a current matching during reconfiguration is at least $M-k$, where M is the size of the original maximum matching and k is the number of current failures. The size of a current matching increases monotonically during reconfiguration. The worst-case reconfiguration time is $O(k \min(|A|,|B|))$ after k failures. But the average-case reconfiguration time is much better. Experimental results were presented, which showed that on the average only several clock ticks are needed for up to 40 failures. Simulation results were also presented for the case when failures are Poisson distributed. These results showed that the algorithm maintains a large-sized current matching.

References

[1] B. Awerbuch, "Complexity of network syncronization," *J.Assoc. Comput. Mach*, vol. 32, pp. 804-823, 1985.

[2] J. W. Greene and A. E. Gamal, "Configuration of VLSI arrays in the presence of defects," *J. Asso. Comp. Mach.*, vol. 31, pp. 694-717, Oct. 1984.

[3] J. E. Hopocroft and R. M. Karp, "An n 2.5 algorithm for maximum matching in bipartite graphs," *SIAM J. Comput.*, vol. 2, pp. 225-231, 1973.

[4] R. M. Karp, U. V. Vazirani, and V. V. Vazirani, "An optimal algorithm for on-line bipartite matching," in *Proc. 22th Annual ACM Symp. Theory of Computing*, 1990, pp. 352-358.

[5] S.-Y. Kuo and W. K. Fuchs, "Spare allocation and reconfiguration in large area VLSI," in *Proc. of 25th ACM/IEEE Design Automation Conference,* 1988, pp 609-612.

[6] S. Micali and V. Vazarina, "An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs," in *Proc. Foundations of Computer Science*, 1980, pp. 17-27.

[7] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*, Prentice-Hall Englewood Cliffs, N.J.,1981.

[8] B. Schieber and S. Moran, "Slowing sequential algorithms for obtaining fast distributed and parallel algorithms: maximum matchings," *Proc. 5th Annual ACM Symp. on Principles of Distributed Computing*, 1986, pp. 282-292.

[9] E. H.-M. Sha and K. Steiglitz, "Reconfigurability and reliability of systolic/wavefront arrays," IEEE Transactions on Computers, To appear.

[10] E. H.-M. Sha and K. Steiglitz, "Explicit constructions for reliable reconfigurable array architectures" in *Proc. Third IEEE Symp. Parallel and Distributed Processing,* Dallas, Texas, Dec. 1991, pp. 640-647.

[11] M. M. Wu and M. C. Loui, " An Efficient Distributed Algorithm for Maximum Matching in General Graphs," *Algorithmica*, vol. 5, 1990, pp. 383-406.

Appendix A: The version A of our algorithm

```
/* Denote by N the node which is performing the following algorithm.  */
/* Let set E be the node set in B which are good, adjacent to N,
and not super nodes.  */
```

Phase 1 for nodes in A

```
  If cur(N) is not good   {
      node N becomes a super node
      Discard any received message }
```

Phase 2

```
  If N is a super node {
      If there exists a free node in E {
      /* Let f be such a free node */
      /* The nodes in the augmenting path should change
      their old matching nodes to be the current matching nodes.*/
      /* Let b be N's old matching node, old(N).  */
2.1         Ask b to send CHANGE_OLD_MATCHING to cur(b)
            /* unmark the alternating path*/
            Set node b to be not reached
2.2         Ask b to send UNMARK_BACKTRACK to all adjacent backtracked nodes
            /* change the old matching node to be the current matching node */
            Set old(N) to be f }
```

```
    Else if there is a node b in E which is unreached {
    /* N can steal this node b */
        Set node b to be reached
2.3     Ask b to send SUPERNODE to cur(b)
        Reset cur(b) and cur(N) }

    Else if old matching node of N is good {
    /* Backtrack from N */
    /* Let b be old(N)*/
2.4     Ask node b to send SUPERNODE to cur(b)
        Set cur(N) to be old(N) }

    Else {
    /* N is temporarily idle*/
        Do nothing }
}
```

Phase 3

```
If N receives SUPERNODE
    Set N to be a super node.

If N receives CHANGE_OLD_MATCHING {
/* Let b be N's old matching node */
3.1     Ask b to send CHANGE_OLD_MATCHING to cur(b)
        Set node b to be not reached
3.2     Ask node node b to send UNMARK_BACKTRACK to all adjacent backtracked nodes
        Set old(N) to be cur(N) }

If N receives UNMARK_BACKTRACK {
/* Let b be N's old matching node */
        Set node b to be not reached
3.3     Ask node b to send UNMARK_BACKTRACK to all adjacent backtracked nodes }
```

For nodes N in B:

```
If old(N) is not good
    Set N to be a super node.
    if old(N) is not the same as cur(N)
        Send SUPERNODE to cur(N).

If N receives CHANGE_OLD_MATCHING and cur(N) is not good
    Set N to be free.
```

Appendix B: Detailed case analyses for lemma 4.2

If the failure at node $b \in B$ happens at the statements in which no message transmissions are involved, we can use the previous Case B1 and Case B2 to analyze the effect. Thus, we only need to consider failures at the statements which have message transmission involving nodes in B. These statements are labeled in Appendix A.

If b fails at the Statement 2.1, b either fails before the message has been sent to b's current matching node, say a', or after. If b fails before sending the message, a' will detect the failure of b in the next clock period and will become a super node, which is similar to the Case B2. Otherwise, if b fails after sending the message, a' will perform the actions for sending the message CHANGE_OLD_MATCHING, and then in the next Phase 1 a' will detect the failure of b, which is similar to Case B1.

When b fails just after 2.2, b does not need to send the unmark messages to all the adjacent backtracked nodes, but it causes no error to send these redundant messages.

If b fails before it sends the message SUPERNODE to its current matching node, say a', at statement 2.3, the current matching node of both a and a', which is b is not good. Thus, in the next Phase 1, a and a' will be super nodes, which is similar to Case B2. Otherwise, if b fails just after it sends the message SUPERNODE, a' will be a super node in Phase 3 and a will be a super node in the next Phase 1, which is also similar to the Case B2. The analysis for the failure of b at 2.4 is similar to the previous analysis for 2.3.

If b fails before it sends the message at statement 3.1, the analysis is the same as we did for statement 2.1. If b fails after sending the message to a', a' will discard this message in the next Phase 1, and becomes the super node, which is similar to the Case B2. The situations when b fails at 3.2, and 3.3 are the same as the one for 2.2.