A Parallel Iterative Improvement Stable Matching Algorithm

Enyue Lu and S.Q. Zheng

Department of Computer Science, University of Texas at Dallas
Richardson, TX 75083-0688, USA
{enyue, sizheng}@utdallas.edu

Abstract. In this paper, we propose a new approach, parallel iterative improvement (PII), to solving the stable matching problem. This approach treats the stable matching problem as an optimization problem with all possible matchings forming its solution space. Since a stable matching always exists for any stable matching problem instance, finding a stable matching is equivalent to finding a matching with the minimum number (which is always zero) of unstable pairs. A particular PII algorithm is presented to show the effectiveness of this approach by constructing a new matching from an existing matching and using techniques such as randomization and greedy selection to speedup the convergence process. Simulation results show that the PII algorithm has better average performance compared with the classical stable matching algorithms and converges in n iterations with high probability. The proposed algorithm is also useful for some real-time applications with stringent time constraint.

1 Introduction

The stable matching problem (or stable marriage problem) was first introduced by Gale and Shapley [6]. Given n men, n women, and $2n$ ranking lists in which each person ranks all members of the opposite sex in order of preference, a *matching* is a set of n pairs of man and woman with each person in exactly one pair. A matching is *unstable* if there are two persons who are not matched with each other, and each of whom strictly prefers the other to his/her partner in the matching; otherwise, the matching is *stable*. Gale and Shapley showed that every instance of the stable matching problem admits at least one stable matching and such a matching can be computed in $O(n^2)$ iterations. The paper of Gale and Shapley sparked much interest in many aspects and variants of the classical stable matching problem. For a good survey on this subject, refer to [8].

Recently, the solutions to the stable matching problem have been applied to switch scheduling for packet/cell switches. Many scheduling algorithms based on stable matchings have been proposed for both input queued (IQ) switches and combined input and output queued (CIOQ) switches (e.g. [3,15,17,18,20]). It has

T.M. Pinkston and V.K. Prasanna (Eds.): HiPC 2003, LNCS 2913, pp. 55–65, 2003.

been shown that scheduling algorithms based on stable matchings can provide QoS guarantees.

For real-time applications, the algorithm proposed by Gale and Shapley, simply GS algorithm, is not fast enough. Attempting to find stable matching algorithms with low complexity was made by many researchers (e.g. [1,7,10,11, 16,21,22]). Up to date, the best known algorithm for stable matching problem takes $O(\sqrt{n} \cdot \log^3 n)$ time [5]. This parallel algorithm runs on a CRCW PRAM (concurrent-read concurrent-write parallel random access machine) of n^4 processors, which makes it infeasible for applications in packet/cell-switched networks. The parallelizability of the stable matching problem is far from being fully understood. It was suggested that parallel stable matching algorithms cannot be expected to provide high speedup on the average [13,19]. Thus, designing efficient parallel algorithms that perform well for most cases is a challenging endeavor.

In this paper, we propose a new approach, parallel iterative improvement (PII), to solving the stable matching problem. The PII algorithm consists of two alternating phases, Initiation_Phase and Iteration_Phase. An Initiation_Phase is a procedure that randomly generates a matching. An Iteration_Phase consists of multiple improvement iterations. We try to speedup the convergence process by exploring parallelism in identifying a subset of unmatched pairs to replace matched pairs for an existing matching so that the number of unstable pairs in newly obtained matching can be reduced. We show that an Initiation_Phase and an iteration of an Iteration_Phase take $O(\log n)$ time on both completely connected multiprocessor system and array with multiple broadcasting buses, and $O(\log^2 n)$ time on both hypercube and MOT, all assumed having n^2 processor elements (PEs). Simulation results show that the PII algorithm has better average performance compared with the classical stable matching algorithms and converges in n iterations with high probability.

2 Preliminaries

Let $M = \{m_1, m_2, \cdots, m_n\}$ and $W = \{w_1, w_2, \cdots, w_n\}$ be the sets of n men and n women respectively. Let $mL_i = \{wr_{i,1}, wr_{i,2}, \cdots, wr_{i,n}\}$ and $wL_i = \{mr_{i,1}, mr_{i,2}, \cdots, mr_{i,n}\}$ be the *ranking lists* for man m_i and woman w_i respectively, where $wr_{i,j}$ (resp. $mr_{i,j}$) is the rank of woman w_j (resp. man m_j) by man m_i (resp. woman w_i). Let A be a *ranking matrix* of size of $n \times n$, where each entry of A is a pair $a_{i,j} = (wr_{i,j}, mr_{j,i})$. We call $wr_{i,j}$ (resp. $mr_{j,i}$) the *left value* (resp. *right value*) of $a_{i,j}$, and denote it by $a^L_{i,j}$ (resp. $a^R_{i,j}$). For convenience, we use $(a^x_{i,j}, a^y_{i,j})$ to denote the indices (i, j) of pair $a_{i,j}$. Clearly, the ordered list of left values of all pairs in row i of A is man ranking list mL_i and the ordered list of right values of all pairs in column j is woman ranking list wL_j. Example 1 shows the ranking matrix obtained from the given ranking lists.

Example 1. An instance of stable matching problem:

Man ranking lists:	Woman ranking lists:	Ranking matrix:
$mL_1 : \{4, 2, 3, 1\};$	$wL_1 : \{1, 4, 2, 3\};$	$4,1 \quad 2,1 \quad 3,4 \quad 1,3$
$mL_2 : \{3, 1, 2, 4\};$	$wL_2 : \{1, 2, 3, 4\};$	$3,4 \quad 1,2 \quad 2,2 \quad 4,1$
$mL_3 : \{2, 4, 1, 3\};$	$wL_3 : \{4, 2, 3, 1\};$	$2,2 \quad 4,3 \quad 1,3 \quad 3,4$
$mL_4 : \{1, 4, 3, 2\}.$	$wL_4 : \{3, 1, 4, 2\}.$	$1,3 \quad 4,4 \quad 3,1 \quad 2,2$

A pair $a_{i,j}$ in A corresponds to a man-woman pair (m_i, w_j). A matching, denoted as $\mathcal{M}$, corresponds to n pairs of A with no two pairs in the same row/column. If a pair of A is in $\mathcal{M}$, it is called a *matching pair* of $\mathcal{M}$ and otherwise a *non-matching pair*. For any matching $\mathcal{M}$ of ranking matrix A, we define the *marked ranking matrix*, $A_{\mathcal{M}}$, as the ranking matrix with all matching pairs marked. Thus for any matching $\mathcal{M}$, each row i (resp. column j) of $A_{\mathcal{M}}$ has exactly one matching pair, which is denoted as $\mathcal{M}(R_i)$ (resp. $\mathcal{M}(C_j)$). A pair $a_{i,j}$ is an *unstable pair* if $a^L_{i,j} < \mathcal{M}(R_i)^L$ and $a^R_{i,j} < \mathcal{M}(C_j)^R$. By the definition of stable matching, we have:

Property 1. A matching $\mathcal{M}$ is stable if and only if there is no unstable pair in $A_{\mathcal{M}}$.

With respect to $A_{\mathcal{M}}$, we define a set NM_1 of *type-1 new matching pairs* (simply *nm_1-pairs*) as follows. If there is no unstable pair in $A_{\mathcal{M}}$, $NM_1 = \emptyset$. Otherwise, for every row R_i with at least one unstable pair, select the one with the minimum left value among all unstable pairs in row R_i as an *nm_1-generating pair*; for every column C_j with at least one nm_1-generating pair, select the one with the minimum right value as an nm_1-pair.

Based on NM_1, we define a set NM_2 of *type-2 new matching pairs* (simply *nm_2-pairs*) by a procedure that first identifies nm_2-generating pairs and then identifies nm_2-pairs using an nm_2-generating graph. For any nm_1-pair $a_{i,j}$ in $A_{\mathcal{M}}$, pair $a_{l,k}$ with $l = \mathcal{M}(C_j)^x$ and $k = \mathcal{M}(R_i)^y$ is called the *nm_2-generating pair* corresponding to $a_{i,j}$. We say that nm_1-pair $a_{i,j}$ and its corresponding nm_2-generating pair $a_{l,k}$ are associated with matching pairs $a_{i,k}$ and $a_{l,j}$. We define an *nm_2-generating graph* $G_{\mathcal{M}}$ as follows: $V(G_{\mathcal{M}}) = \{$all nm_2-generating pairs$\}$, and $E(G_{\mathcal{M}}) = \{e = (u, v)|$ two nm_2-generating pairs u and v are associated with a common matching pair$\}$. Since each nm_2-generating pair is associated with 2 matching pairs, we have:

Property 2. Given any $A_{\mathcal{M}}$, the degree of nm_2-generating graph $G_{\mathcal{M}}$ is at most 2.

By Property 2, each connected component in $G_{\mathcal{M}}$ is a cycle or chain, named *nm_2-generating cycle or nm_2-generating chain* (an isolated node is a chain of length 0). If a node in $G_{\mathcal{M}}$ has degree 2, it is called an *internal node* and otherwise an *end node*. Clearly, if an nm_2-generating pair $a_{i,j}$ is an internal node in $G_{\mathcal{M}}$, there are two nm_1-pairs, one in row i and the other in column j; if an nm_2-generating pair $a_{i,j}$ is an end node in $G_{\mathcal{M}}$, there is at most one nm_1-pair in

row i or column j. We call an end node $a_{i,j}$ a *row end* (resp. *column end*) of an nm_2-generating chain if there is no nm_1-pair in row i (resp. column j) of $A_{\mathcal{M}}$. An isolated node is both row end and column end.

By the nm_2-generating graph, we can generate the set NM_2 of nm_2-pairs as follows. For each nm_2-generating chain with row end a_{i_1,j_1} and column end a_{i_2,j_2}, we generate an nm_2-pair a_{i_1,j_2}. No nm_2-pair is generated from any nm_2-generating cycle. Hence, there is a one-to-one correspondence between an nm_2-generating chain and an nm_2-pair. Let $NM = NM_1 \cup NM_2$. We call NM the set of *new matching pairs* (simply *nm-pairs*). Based on the way that NM is generated, we know that NM_1 and NM_2 are disjoint, and each row/column of $A_{\mathcal{M}}$ contains at most one nm-pair.

A matching pair $a_{i,j}$ in $A_{\mathcal{M}}$ is called a *replaced matching pair* (simply *rm-pair*), if it is in the same row/column of an nm-pair. We denote the set of rm-pairs by RM. Based on the way that RM is constructed, we have:

Lemma 1. *If there is at least one unstable pair in $A_{\mathcal{M}}$, then $\mathcal{M}' = (\mathcal{M} - RM) \cup NM$ is a matching different from $\mathcal{M}$.*

3 Parallel Iterative Improvement Matching Algorithm

In this section, we present our main result, a *parallel iterative improvement algorithm* (PII algorithm) for a completely connected multiprocessor system, which consists of a set of PEs connected in such a way that there is a direct connection between every pair of PEs. We assume that each PE can communicate with at most one adjacent PE during every communication step. The PII algorithm uses n^2 PEs. To facilitate our discussion, these n^2 PEs are placed as an $n \times n$ array. As input, $PE_{i,j}$, $(1 \leq i, j \leq n)$, contains $a_{i,j}$ of ranking matrix A. When the algorithm terminates, a stable matching is found by $PE_{i,j}$ indicating whether pair (m_i, w_j) is in the matching. The key idea of the PII algorithm is to construct a new matching $\mathcal{M}'$ from an existing matching $\mathcal{M}$ in hope that $\mathcal{M}'$ is "closer" to a stable matching than $\mathcal{M}$.

3.1 Constructing an Initial Matching

To randomly generate an initial matching can be reduced to generate a random permutation, which can be done by a sequential algorithm proposed in [4]. In the following, we show how to implement it in parallel on a completely connected multiprocessor system with N^2 PEs.

Let each PE maintain a pointer. Initially, every $PE_{i,j}$ sets its pointer to point to $PE_{i+1,j}$, $(1 \leq i \leq n-1)$, and as a result, there are n disjoint lists. Then, each $PE_{i,i}$ will randomly choose a j $(i \leq j \leq n)$ to swap their pointers, i.e. $PE_{i,i}$ points to $PE_{i+1,j}$ and $PE_{i,j}$ points to $PE_{i+1,i}$. Consequently, n new disjointed lists originated from $PE_{1,j}$ are formed. After performing $\log n$ times of pointer jumping [12], each $PE_{1,j}$ finds another end $PE_{n,p(1,j)}$ of its list, where $p(1,j)$ is the column position of the PE pointed by $PE_{1,j}$. Hence, a matching $\{(j, p(1,j)) | 1 \leq j \leq n\}$ is formed. Clearly, this parallel implementation takes $O(\log n)$ time since each list has length of n.

3.2 Construct a New Matching from an Existing Matching

A basic operation of PII algorithm is to construct a new matching $\mathcal{M}' = (\mathcal{M} - RM) \cup NM$ from an existing matching $\mathcal{M}$ if $\mathcal{M}$ is unstable. In the following, we describe 6 steps to carry out this operation.

Step 1: Recognize Unstable Pairs. Every PE with a matching pair in $\mathcal{M}$ broadcasts its column/row position and the left/right value of its matching pair to all PEs in the same row/column. If $PE_{i,j}$'s both values are smaller, set its Boolean variable $u_{i,j} := true$, which indicates that pair $a_{i,j}$ is unstable; otherwise set $u_{i,j} := false$. The broadcasting in rows/columns takes $O(\log n)$ time.

Step 2: Stability Checking. Find if there exists a $PE_{i,j}$ with $u_{i,j} := true$ by binary searching in rows/columns. Since each row/column has n PEs, the searching takes $O(\log n)$ time. If $f_{i,j} := false$ for any $PE_{i,j}$, then the current matching $\mathcal{M}$ is stable, and the algorithm terminates. Otherwise, go to the next step.

Step 3: Find NM_1. For each row with at least one unstable pair, find the unstable pair with the minimum left value, and mark this pair as an nm_1-generating pair. For each column with at least one nm_1-generating pair, find the nm_1-generating pair with the minimum right value, and mark this pair as an nm_1-pair. The find-minimum operation in rows/columns takes $O(\log n)$ time.

Step 4: Find nm_2-Generating Pairs. For each $PE_{i,j}$ containing an nm_1-pair, mark the pair in $PE_{l,k}$ as a nm_2-generating pair, where $l = \mathcal{M}(C_j)^x$ and $k = \mathcal{M}(R_i)^y$. Clearly, this step only takes $O(1)$ time.

Step 5: Find NM_2. This step consists of two major objectives: (1) each nm_2-generating node that is both row end and column end in $G_{\mathcal{M}}$ recognizes itself as an isolated node, and (2) the row end of each nm_2-generating chain in $G_{\mathcal{M}}$ finds its column end. Let each $PE_{i,j}$ containing an nm_2-generating pair maintain two pointers, r-pointer and c-pointer. The r-pointer (resp. c-pointer) of $PE_{i,j}$ points to the PE containing an nm_2-generating pair in column $\mathcal{M}(R_i)^y$ (resp. row $\mathcal{M}(C_j)^x$) if there is an nm_1-pair in row i (resp. column j), and otherwise to itself. If both r-pointer and c-pointer of $PE_{i,j}$ point to itself, then it corresponds to an isolated node in $G_{\mathcal{M}}$; if the r-pointer (resp. c-pointer) of $PE_{i,j}$ points to itself but another pointer points to some other PE, then $PE_{i,j}$ contains an nm_2-generating pair that is the row (resp. column) end of an nm_2-generating chain; if both r-pointer and c-pointer of $PE_{i,j}$ point to other PEs, its nm_2-generating pair corresponds to an internal node of $G_{\mathcal{M}}$. Fig. 1 shows an example. By a completely connected multiprocessor with N^2 PEs, objective (1) can be easily achieved in $O(1)$ time and objective (2) can be achieved by performing $\lceil \log n \rceil$ times of pointer jumping [12] since the length of each nm_2-generating chain is at most n. Once objectives (1) and (2) are accomplished, the nm_2-pairs can be easily computed in $O(1)$ time.

Step 6: Construct New Matching. Each $PE_{i,j}$ containing an nm-pair marks the matching pair in row i as a replaced pair, and marks itself as a matching pair. This step takes $O(1)$ time.

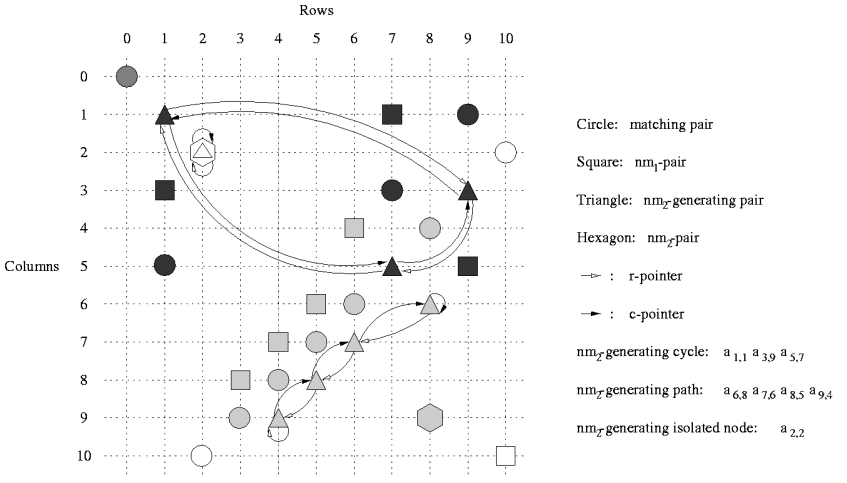

Fig. 1. Finding a new matching $\mathcal{M}'$ from an existing matching $\mathcal{M}$, where $\mathcal{M} = \{a_{0,0}, a_{1,9}, a_{2,10}, a_{3,7}, a_{4,8}, a_{5,1}, a_{6,6}, a_{7,5}, a_{8,4}, a_{9,3}, a_{10,2}\}$, $NM_1 = \{a_{1,7}, a_{3,1}, a_{4,6}, a_{5,9}, a_{6,5}, a_{7,4}, a_{8,3}, a_{10,10}\}$, $NM_2 = \{a_{2,2}, a_{9,8}\}$, $RM = \{a_{1,9}, a_{2,10}, a_{3,7}, a_{4,8}, a_{5,1}, a_{6,6}, a_{7,5}, a_{8,4}, a_{9,3}, a_{10,2}\}$ and $\mathcal{M}' = (\mathcal{M} - RM) \cup NM = \{a_{0,0}\} \cup NM_1 \cup NM_2$.

3.3 PII Algorithm

Now we are ready to present our PII algorithm. Conceptually, the PII algorithm has two alternating phases: INITIATION_PHASE and ITERATION_PHASE. The INITIATION_PHASE finds an initial matching arbitrarily. The ITERATION_PHASE contains at most $c \cdot n$ iterations, where c is a constant for controlling the number of iterations in the ITERATION_PHASE. Each iteration of ITERATION_PHASE checks whether an existing matching $\mathcal{M}$ is stable. If $\mathcal{M}$ is stable, the algorithm terminates; otherwise, a new matching $\mathcal{M}'$ is constructed. Then, $\mathcal{M}'$ is used as $\mathcal{M}$ for the next iteration. After $c \cdot n$ iterations in each ITERATION_PHASE, the PII algorithm goes back to INITIATION_PHASE to generate a new initial matching randomly and a new ITERATION_PHASE is effected based on this new generated matching. As we analyzed, an INITIATION_PHASE and an iteration of an ITERATION_PHASE takes $O(\log n)$ time on a completely connected multiprocessor system with n^2 PEs.

In an iteration of an ITERATION_PHASE, a new matching $\mathcal{M}' = (\mathcal{M} - RM) \cup NM_1 \cup NM_2$ is constructed from an existing matching $\mathcal{M}$. It is easy to verify that the pairs in NM_1 were unstable for $\mathcal{M}$, but become stable for $\mathcal{M}'$; the pairs in NM_2 are stable for $\mathcal{M}'$, regardless whether they were stable for $\mathcal{M}$; and the pairs in $\mathcal{M}$, which were stable for $\mathcal{M}$, remain to be stable for $\mathcal{M}'$. Intuitively, the number of unstable pairs for $\mathcal{M}'$ is smaller than the number of unstable pairs for $\mathcal{M}$. For most cases, it is true. This is the heuristic behind the PII algorithm.

However, new unstable pairs may be generated for $\mathcal{M}'$. Let the initial matching be $\mathcal{M}_0$ and the matching generated in the i-th iteration be $\mathcal{M}_i$. Since the

set of nm_1-pairs, nm_2-pairs and rm-pairs with respect to $\mathcal{M}_{i-1}$ is unique, the matching $\mathcal{M}_i$ is constructed uniquely from $\mathcal{M}_{i-1}$. Hence, if $\mathcal{M}_i \in \{\mathcal{M}_j | j \in \{0, 1, \cdots, i-1\}\}$, i.e. the newly generated matching is the same as a previously generated matching, no stable matching can be found. It is possible to include a procedure for detecting this cyclic situation. Such a procedure, however, is too time-consuming. This is why we decided to start a new round after $c \cdot n$ iterations of an ITERATION_PHASE. The random permutation generating algorithm we used generates random matchings with uniform distribution according to [2]. Therefore, by the existence of a stable matching, the PII algorithm can always find one for any instance of stable matching problem.

Our simulation results (see Section 5) indicate that the PII algorithm has better performance compared with GS algorithm. However, we are unable to theoretically exclude the possibility that the cases in which the total number of iterations is very large. In order to enforce a bound for the number of iterations, we propose to run the PII algorithm and parallel GS algorithm simultaneously in a time-sharing fashion. We denote this modified PII algorithm as PII-GS, which terminates once one of the algorithms generates a stable matching. Clearly, the PII-GS algorithm converges to a stable matching with $O(n^2)$ iterations in the worst case.

4 Implementations of PII Algorithm on Parallel Computing Machine Models

In this section, we consider implementing the PII algorithm on three well-known parallel computing systems − hypercube, mesh of trees (MOT) and array with multiple broadcasting buses. Without loss of generality, assume $n = 2^k$. If $2^k < n < 2^{k+1}$, the PII algorithm can be implemented on a 2^{2k}-processor system with a constant slow-down factor. The n^2 PEs in each system are placed as $n \times n$ array, and n PEs in each row/column form a row/column connection (see Fig. 2). We assume that our parallel computing systems operate in a synchronous fashion. Basic $O(1)$-time parallel operations of a hypercube and a MOT can be found in [14]. For an array with multiple broadcasting buses, we assume that each bus has a controller. A processor can request to communicate with the controller or any other processor on the bus. At any time, more than one processor on a bus may send requests to the bus controller, and the controller selects one request (if any) to grant the bus access arbitrarily. The controller of a bus can broadcast a message to all the processors on the bus. We assume that each processor-to-processor, processor-to-controller and broadcasting operation takes $O(1)$ time.

It is simple to notice that multiple-broadcasting, finding minimum, and pointer jumping are the most time consuming operations in the PII algorithm. The pointer jumping can be carried out by sorting. Let C be a parallel computing machine with n^2 processors, and let $T_B(n)$, $T_M(n)$ and $T_S(n)$ be the time required for multiple-broadcasting, finding minimum and sorting on C, respectively. Then, an INITIATION_PHASE of PII algorithm can be implemented

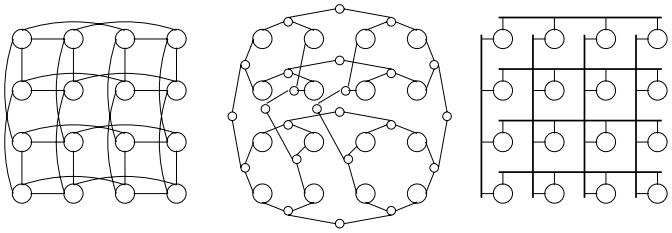

Fig. 2. (a) A 16-processor hypercube (b) A 4×4 mesh of trees (c) A 4×4 array with multiple broadcasting buses

on C in $O(T_S(n) \cdot \log n)$ time, and each iteration of an ITERATION_PHASE of PII algorithm can be implemented on C in $O(\max\{T_B(n), T_M(n), T_S(n) \cdot \log n\})$ time.

For a hypercube and a MOT, the operations of broadcasting and finding-minimum in PII are performed in parallel row-wise or column-wise. Thus, $T_B(n) = T_M(n) = O(\log n)$ for a hypercube and a MOT. For an array with multiple broadcasting buses, $T_B(n) = O(1)$. Finding-minimum operation can be carried out on a bus in $O(\log n)$ time using a binary searching method. For an n^2-processor hypercube $T_S(n) = O(\log^2 n)$ while $T_S(n) = \Omega(n)$ for a MOT and an array with multiple broadcasting buses since either of their bisection widths is n. If we use sorting to implement pointer jumping operations, both an INITIATION_PHASE and an iteration in an ITERATION_PHASE of PII algorithm require $O(\log^3 n)$ time on a hypercube and $\Omega(n \log n)$ time on a MOT and an array with multiple broadcasting buses. In the following, however, we show that sorting can be avoided on these parallel computing models using special features of PII algorithm.

First, we show how to implement an INITIATION_PHASE without pointer jumping. This can be done by adopting a parallel implementation in [9] of the algorithm of [4]. Let π_i, $(1 \leq i \leq n-1)$, be the permutation interchanging i and r_i that is chosen randomly from the set $\{i, \cdots, n\}$ while leaving other elements of $\{1, 2, \cdots, n\}$ fixed. Let π_n be an identity permutation. Initially, we use row i to represent π_i. The computation $\pi = \pi_1 \circ \pi_2 \circ \cdots \circ \pi_{n-1} \circ \pi_n$ is organized in a complete binary tree of height $\log n$. For example, for $n = 8$, $\pi = ((\pi_1 \circ \pi_2) \circ (\pi_3 \circ \pi_4)) \circ ((\pi_5 \circ \pi_6) \circ (\pi_7 \circ \pi_8))$. Hence, all that remains is to consider the composition of two permutations. Given a permutation π', let $D(\pi') = \{i | 1 \leq i \leq n, \text{ and } \pi'(i) \neq i\}$. The algorithm of [9] associates $|D(\pi')|$ processors to π'. In our implementation, we mimic the operations of one processor in [9] using a set of processors and their connections. More specifically, we associate each row/column i to π_i at the beginning. Let $\pi_{(i,j)} = \pi_i \circ \pi_{i+1} \circ \cdots \circ \pi_j = \pi_{(i,(j-i+1)/2)} \circ \pi_{((j-i+1)/2+1,j)}$, where $j-i+1$ is an integer of power of 2. Note that $|D(\pi_{(i,j)})| \leq j - i + 1$. Thus, we can use row i through row j and column i through column j to perform the operations assigned to the processors for

computing $\pi_{(i,j)}$ in the algorithm of [9]. The communication paths for computing the composition of permutations at the same level of the binary computation tree are disjoint, because they use disjoint sets of row and column connections. Since the height of the binary computation tree is $O(\log n)$, an INITIATION_PHASE of PII algorithm takes $O(\log^2 n)$ time on a hypercube and a MOT. If an array with multiple broadcasting buses is used, an INITIATION_PHASE of PII algorithm takes $O(\log n)$ time.

We now show how to implement an iteration of ITERATION_PHASE without using pointer jumping. Since each row/column contains at most one nm_2-generating pair, each pointer jumping of Step 5 can be decomposed into disjoint parallel 1-to-1 row communications followed by disjoint parallel 1-to-1 column communications without conflicts. Thus, every pointer jumping can be implemented in $O(\log^2 n)$ time on a hypercube and a MOT, and in $O(\log n)$ time on an array with multiple broadcasting buses. We also note that simulating an $n \times n$ MOT by an $n/2 \times n/2$ MOT (which has $3n^2/4 - n < n^2$ processors) results in a constant slowdown factor. To summarize, we show the improvement of the time complexity of PII algorithm on three parallel computing systems in Table 1.

Table 1. Time complexity for implementations of PII algorithm on three parallel computing machine models

Machine models	INITIATION_PHASE		An iteration in ITERATION_PHASE	
	with sorting	without sorting	with sorting	without sorting
Hypercube	$O(\log^3 n)$	$O(\log^2 n)$	$O(\log^3 n)$	$O(\log^2 n)$
MOT	$O(n \log n)$	$O(\log^2 n)$	$O(n \log n)$	$O(\log^2 n)$
Array with Buses	$O(n \log n)$	$O(\log n)$	$O(n \log n)$	$O(\log n)$

5 Simulation Results

We have simulated PII, PII-GS, and parallel GS algorithms for different sizes $n \in \{10, 20, \cdots, 100\}$ of stable matching, with 10000 runs each. The ranking lists and initial matchings are generated by random permutation algorithm [4]. Each ITERATION_PHASE contains n iterations. The performance comparisons are based on the average number of parallel iterations for each algorithm to generate a stable matching and the frequency for each algorithm to converge in n iterations. From the simulation, we notice that PII and PII-GS algorithms significantly outperform GS algorithm. Fig. 3 shows that PII and PII-GS algorithms converge in n iterations with very high probabilities, while the probability for GS algorithm to converge with the same number of iterations decreases quickly as the sizes of problem increase.

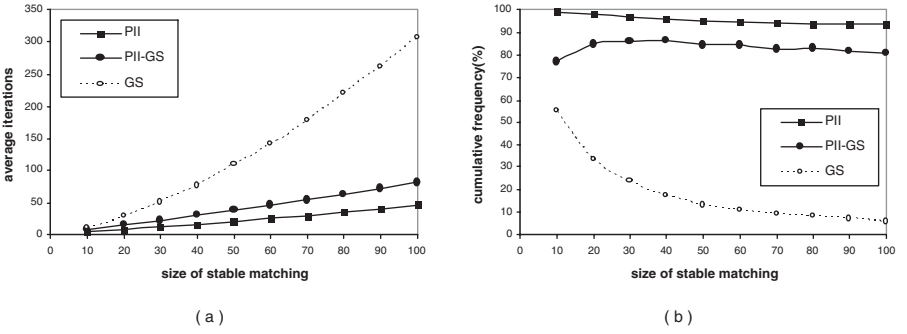

Fig. 3. Performance Comparisons (a) The average number of iterations for algorithms to find a stable matching (b) The frequencies for algorithms to find a stable matching within n iterations

6 Concluding Remarks

In this paper, we proposed a new approach, parallel iterative improvement, to solving the stable matching problem. The PII algorithm requires n^2 PEs, among which n PEs are required to perform arithmetic operations (for random number generation) and the other PEs can be simple comparators. The classical GS algorithm and most existing stable matching algorithms can only find the man-optimal or woman-optimal stable matching. By [8], the man (resp. woman)-optimal stable matching is women (resp. men)-pessimal, i.e. every man/woman gets the best partner while every woman/man gets the worst partner over all stable matchings. However, due to randomness, the PII algorithm constructs a stable matching that is contained in the set of stable matchings. Therefore, this algorithm will generate the stable matching with more fairness.

In some applications, such as real-time packet/cell scheduling for a switch, stable matching is desirable, but may not be found quickly within tight time constraint. Thus, finding a "near-stable" matching by relaxing solution quality to satisfy time constraint is more important for such applications. Most of existing parallel stable matching algorithms cannot guarantee a matching with a small number of unstable pairs within a given time interval. Interrupting the computation of such an algorithm does not result in any matching. However, the PII algorithm can be stopped at any time. By maintaining the matching with the minimum number of unstable pairs found so far, a matching that is close to a stable matching can be computed quickly.

References

1. Abeledo, H., Rothblum, U.G.: "Paths to marriage stability", *Discrete Applied Mathematics*, 63 (1995) 1–12

2. Anderson, R: "Parallel algorithms for generating random permutations on a shared memory machine", *Proc. of the 2nd ACM Symposium on Parallel Algorithms and Architectures*, (1990) 95–102
3. Chuang, S.T., Goel, A. , McKeown, N., Prabhakar, B.: "Matching output queuing with a combined input/output-queued switch", *IEEE Journal on Selected Areas in Communications*, Vol. 17, No. 6 (1999) 1030–1039
4. Durstenfeld, R.: "Random permutation (Algorithm 235)", *Communication of ACM*, Vol. 7, No. 7 (1964) 420
5. Feder, T., Megiddo, N., Plotkin, S.: "A sublinear parallel algorithm for stable matching", *Theoretical Computer Science*, Vol. 233 (2000) 297–308
6. Gale, D., Shapley, L.S.: "College admissions and the stability of marriage", *American Mathematical Monthly*, Vol. 69 (1962) 9–15
7. Gusfield, D.: "Three fast algorithms for four problems in stable marriage", *SIAM Journal on Computing*, Vol. 16, No. 1 (1987) 111–128
8. Gusfield, D., Irving, R.W.: *The Stable Marriage Problem Structure and Algorithms*, MIT Press (1989)
9. Hagerup, T.: "Fast parallel generation of random permutations", *Proc. of the 18th Annual International Colloquium on Automata, Languages and Programming*, (1991) 405–416
10. Hattori,T., Yamasaki,T., Kumano, M.: "New fast iteration algorithm for the solution of generalized stable marriage problem", *Proc. of IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 6 (1999) 1051 -1056
11. Hull, M.E.C.: "A parallel view of stable marriages", *Information Processing Letters*, Vol. 18, No. 1 (1984) 63–66
12. Jaja, J.: *An Introduction to Parallel Algorithms*, Addison-Wesley (1992)
13. Kapur, D., Krishnamoorthy, M.S.: "Worst-case choice for the stable marriage problem", *Information Processing Letters*, Vol. 21 (1985) 27–30
14. Leighton, F. T.: *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*, Morgan Kaufmann Publishers (1992)
15. McKeown, N.: "Scheduling algorithms for input-buffered cell switches", Ph.D. Thesis, University of California at Berkeley (1995)
16. McVitie, D.G., Wilson, L.B.: "The stable marriage problem", *Communication of the ACM*, Vol. 14, No. 7 (1971) 486–490
17. Nong, G., Hamdi, M.: "On the provision of quality-of-service guarantees for input queued switches", *IEEE Communications Magazine*, Vol. 38, No. 12 (2000) 62–69
18. Prabhakar, B., McKeown, N.: "On the speedup required for combined input- and output-queued switching", *Automatica*, Vol. 35, No. 12 (1999) 1909–1920
19. Quinn, M.J.: "A note on two parallel algorithms to solve the stable marriage problem", *BIT*, Vol. 25 (1985) 473–476
20. Stoica, I., Zhang, H.: "Exact emulation of an output queuing switch by a combined input output queuing switch", *Proc. of the 6th IEEE/IFIP IWQoS'98*, Napa Valley, CA, (1998) 218–224
21. Subramanian, A.: "A new approach to stable matching problems", *SIAM Journal on Computing*, Vol. 23, No. 4 (1994) 671–700
22. Tseng, S.S., Lee, R.C.T.: "A parallel algorithm to solve the stable marriage algorithm", *BIT*, Vol. 24 (1984) 308–316