AN EFFICIENT PARALLEL ALGORITHM FOR
THE ASSIGNMENT PROBLEM ON THE PLANE*

Constantine N.K. Osiakwan
Bell Northern Research
3500 Carling Avenue
Ottawa, Ontario, K1Y 4H7

Selim G. Akl
Computing and Information Science
Queen's University
Kingston, Ontario, K7L 3N6

Abstract
We consider a parallel algorithm for minimum weight perfect
matching on complete bipartite graphs induced by two
disjoint sets of points on the plane. The algorithm
is designed for the EREW PRAM or EREW RPRAM model of
parallel computation with p (a function of n) processors.
For points on the Euclidean plane, the algorithm executes
in \(O(n^3/p^2 + n^2 \log n + n^{2.5} \log n/p)\) time for \(p < \sqrt{n}\). We
achieve a perfect speedup algorithm when \(p = \frac{n}{\log n}\) and \(\sqrt{n}/p\) is not a constant.

1 Introduction
Let \(G = (U, V)\) be a complete bipartite graph on the
plane induced by two sets of points \(U\) and \(V\), with
\(|U| = |V| = n\). A perfect matching \(M\) of a bipar-
tite graph \(G\) on the plane is a pairing \((u_i, v_j)\) of the
vertices in \(U\) with those in \(V\) such that every vertex
in \(U\) is paired with exactly one vertex in \(V\) and vice
versa. A minimum weight perfect matching is a perfect
matching \(M\) of \(G\) such that the sum of the distances
between the pairs \(u_i\) and \(v_j\) is minimum over all perfect
matchings. An assignment problem is to determine a
minimum weight perfect matching of \(G\).

An \(O(n^3)\) algorithm for the assignment problem,
the Hungarian method, was designed by Kuhn [10].
Other algorithms have been presented [8, 13]. On com-
plete bipartite graphs, the time complexities of these
other implementations are not better than the original
Hungarian method. Faster but sub-optimal heuristics
for matching on the plane have appeared in the litera-
ture, for example [2]. Recently, Vaidya [14] improved
the time complexity of the Hungarian method for com-
plete bipartite graphs induced by points on the plane.
He obtained an \(O(n^{2.5} \log n)\) time algorithm, when the
distance is Euclidean or Manhattan.

One model of parallel computation considered in
this paper is the real parallel random access machine
(RPRAM). This model is the same as the parallel ran-
don access machine PRAM model of parallel com-
putation, except that we assume that square roots can be
computed in constant time. The other model of paral-
lel computation we use is the PRAM. On the PRAM,
we assume that distances requiring square root com-
putations have already been computed and are avail-
able at the start of the algorithm.

Parallel algorithms for the assignment problem
have been designed, [7, 9, 3]. For a complete bipartite
graph on the plane, these algorithms do not achieve op-
timal speedup with respect to the algorithm of Vaidya.
In this paper we present a parallel algorithm for the
assignment problem on a complete weighted bipar-
tite graph on the plane. The algorithm executes in
\(O(n^3/p^2 + n^2 \log n + n^{2.5} \log n/p)\) time, on the EREW
RPRAM model or the EREW PRAM model using \(p\)
processors where \(p < \sqrt{n}\). It is a parallelization of
the algorithm of Vaidya. The algorithm also achieves
an optimal speedup (see [1]) when \(\sqrt{n}/\log n \leq p\)
and \(\sqrt{n}/p\) is not a constant.

In Section 2 we give a linear program (LP) for-
malization of the assignment problem on the plane. The
improvement due to Vaidya [Vaidya '89a] on which
our parallel algorithm is based is briefly presented in
Section 3. We discuss a parallel algorithm to solve the
assignment problem on the Euclidean plane in Section
4. Section 5 consists of a summary of the paper and
some open problems.

2 An LP Formulation
Let \(d(u_i, v_j)\) be the distance between \(u_i \in U\) and \(v_j \in
V, 1 \leq i, j \leq n\). An LP [10, 11] of the assignment
problem on the plane is:

\[
\begin{align*}
\text{minimize} & \quad \sum_{(u_i, v_j)} d(u_i, v_j) x_{ij} \\
\text{subject to} \quad & \sum_{u_i} x_{ij} = 1 \quad i = 1, \ldots, n \\
& \sum_{v_j} x_{ij} = 1 \quad j = 1, \ldots, n \\
& x_{ij} \geq 0
\end{align*}
\]

where the pairing \((u_i, v_j)\) is in the matching \(M\) if and
only if \(x_{ij} = 1\). To solve this linear program a dual to
the linear program is formulated as

*This research was supported by the Natural Sciences and
Engineering Research Council of Canada under grant A3336.
maximize \( \sum_i \alpha_i + \sum_j \beta_j \)
subject to
\( \alpha_i + \beta_j \leq d(u_i, v_j) \quad 1 \leq i, j \leq n \)
\( \alpha_i, \beta_j \) unconstrained

where \( \alpha_i \) and \( \beta_j \) are the dual variables associated with \( u_i \) and \( v_j \) respectively. Orthogonality conditions that are necessary and sufficient for optimality of the primal and dual solutions are

\[ x_{ij} > 0 \Rightarrow \alpha_i + \beta_j = d(u_i, v_j) \quad (1) \]
\[ \alpha_i \neq 0 \Rightarrow \sum_j x_{ij} = 1 \quad i = 1, \ldots, n \quad (2) \]
\[ \beta_j \neq 0 \Rightarrow \sum_i x_{ij} = 1 \quad j = 1, \ldots, n \quad (3) \]

3 Vaidya's Algorithm in Brief

Let \( \sigma_{ij} = d(u_i, v_j) - \alpha_i - \beta_j \) be the slack of the edge \((u_i, v_j)\). To solve the assignment problem, Vaidya (a) associated a weight \( w(u_i) \) for \( u_i \in U \) and \( w(v_j) \) for \( v_j \in V \) related to the dual variable, with each vertex, and (b) reduced determination of the edge \((u_i, v_j) \in \mathcal{M}\) with minimum slack to be included in an alternating tree, where \( u_i \) is in an alternating tree and \( v_j \) is not, to a geometric (nearest neighbor) query problem that involves the weights. weighted Voronoi diagram (WVD) [5, 4] is used to solve the nearest neighbour problem for points on the Euclidean plane.

Let \( \Theta \) be a forest of alternating trees and a variable \( \Delta \) be used to keep track of the sum of the amount, \( \delta \), by which the dual variables change. The relationship between the weight and dual variable is given as \( \alpha_i = w(u_i) + \Delta, \beta_j = w(v_j) - \Delta \), where \( u_i \) and \( v_j \) are in \( \Theta \). The slack of an edge \((u_i, v_j)\) to be included in \( \Theta \) is then given by \( \sigma_{ij} = d(u_i, v_j) - w(u_i) - w(v_j) - \Delta \). At the beginning of a phase, \( \Delta \) is initialized to 0. When the dual variables are to be revised, \( \delta \) is added to \( \Delta \) instead of revising the dual variables. When a vertex gets included in \( \Theta \), the weight is initialized to the dual variable. At the end of a phase, the matching is augmented and the dual variables are revised using \( \Delta \) and the weights.

4 A Parallel Algorithm

The sweepline technique of Fortune for determining WVD looks inherently sequential, since the boundaries of the next point to be processed depends on the previous ones already inserted into a priority queue. It is, therefore, unlikely that a perfect speedup parallel algorithm for the EREW PRAM that uses \( p \) processors, where \( p \) is a function of the number of points, \( m \), and whose value is at most \( m \), can be designed.

WVD is not dynamic with respect to deletion. Since it is not likely that a perfect speedup parallel algorithm can be designed for constructing WVD, we modify the size of sets for which the data structures are to be reconstructed each time an edge is included in an alternating tree. We reduce the process of recovering an alternating path to a prefix computation. We use \( p (\leq \sqrt{n}) \) processors, subscripted 1 to \( p \).

Theorem 2

The assignment problem on the Euclidean plane can be solved in \( O(n^3/p^2 + n^2 \log n + n^2/\sqrt{n} \log n/p) \) time on the EREW RPRAM or the EREW PRAM with \( p (< \sqrt{n}) \) processors.

4.1 Initialization

We present the proof of Theorem 2 in this section. Determining an augmenting path is begun by rooting alternating trees at each exposed vertex in \( U \) in \( O(n/p) \) time. The set \( S_1 \) is initialized to the set of exposed vertices in \( U \). If \( S_1 \) is empty, then a solution has been found. Otherwise, the set \( S_1 \) is set to \( S_1 \) and the set \( F \) is set to \( V \). Determining shortest[\( v_j, S_1' \)] for all \( v_j \in F \) can be done as follows. First, each processor builds the WVD for \( S_1 \) in its own memory in \( O(\sqrt{n} \log n) \) time. Then we assign at most \( n/p \) vertices \( v_j \) in \( F \) to a processor. Each processor computes shortest[\( v_j, S_1' \)] in \( O(n \log n/p) \) time; \( O((n/p + \sqrt{n}) \log n) \) time for all \( v_j \in F \) and for each \( S_1' \). This is done only once in a phase for each \( S_1' \). Since there will be at most \( \sqrt{n} \) \( S_2 \)'s of size \( \sqrt{n} \) in a phase, we obtain \( O((n\sqrt{n}/p + n) \log n) \) time.

In determining the initial edges shortest[\( v_j, S_1 \)] for all \( v_j \in F \), the set \( S_1' \) is partitioned into at most \( p \) subsets \( S_1'y, 1 \leq y \leq p \), each of size at most \( n/p \). Each processor constructs a WVD for a subset \( S_1'y \) of \( S_1 \). For each \( v_j \in F \), shortest[\( v_j, S_1 \)] is determined as follows. The vertices \( v_j \) in \( F \) are distributed among the processors. Each processor \( P_i \) begins with shortest[\( v_j, S_1 \)] to be shortest[\( v_j, S_1'y \)]. Let this process be done at time step 1. At the \( i^{th} \) time step, processor \( P_i \) determines shortest[\( v_j, S_1 \)] to be the shortest of the edges shortest[\( v_j, S_1 \)] and shortest[\( v_j, S_1'y \)]. Each processor \( P_i \) creates and maintains a priority queue \( PQ_i \) from the edges shortest[\( v_j, S_1 \)] of the vertices \( v_j \) it is assigned. The time to determine shortest[\( v_j, S_1 \)] is \( O(n \log n/p \times p) \) or \( O(n \log n) \), and this is done once in a phase.

Instead of constructing one WVD for the set \( S_1 \), each time \( S_2 \) is added to \( S_1 \), WVD's will be constructed for subsets \( S_1' \) of \( S_1 \). Each subset \( S_1' \), of size at most \( \sqrt{n} \), will be the set \( S_2 \) added to \( S_1 \). Initially, we let \( S_1' \) be the set of exposed vertices in \( U \). At the beginning of the algorithm, the edges shortest[\( v_j, S_1 \)] are determined. After the first set of edges shortest[\( v_j, S_1 \)] have been obtained for each \( v_j \in F \), determining subsequent shortest[\( v_j, S_1 \)], is done by determining the shortest of the edges shortest[\( v_j, S_1'y \)] and shortest[\( v_j, S_1' \)], \( 2 \leq y \leq \sqrt{n} \), where \( S_1'y \) is the set \( S_2 \) whose size has become \( \sqrt{n} \) and has been added to \( S_1 \).
Next we preprocess the vertices in $F$ so that for a vertex $u_i \notin S$, being inserted in $S_2$, shortest$[u_i, F]$ can be determined very fast. Initial WVD's for the $F_i$'s can be constructed in $O(\sqrt{n \log n} \times \sqrt{n/p})$ time. Also WVD's are constructed for each subsubset $F_{tk}$, in the same time. All processors set $w(u_i)$ to $\beta_1$ for each $u_i \in S$ and $w(v_j)$ to $\beta_2$ for each $v_j \in F$.

In growing alternating trees, an edge $(u_i, v_j)$ for which
\[
s = d(u_i, v_j) - w(u_i) - w(v_j)
\]
is determined from the priority queues $PQ_1$ and $PQ_2$. Each processor determines the edge with minimum slack in its priority queue. The edge with overall minimum slack $s$ is then determined by all processors. The edge $(u_i, v_j)$, together with the slack $s$, are made known to all processors. If $s - \Delta$ is greater than zero, $\Delta$ is incremented by $s - \Delta$. However if $s - \Delta$ is zero and the vertex $v_j$ is exposed, and an augmenting path has been discovered. The augmenting path is recovered using lemma 1.

**Lemma 1** [12]

Tracing an augmenting path and augmenting a matching in a bipartite graph can be done in $O(n \log n/p + \log^2 n)$ time on the EREW PRAM using $p$ processors. □

If the vertex $v_j$ is not exposed and $s - \Delta$ is zero, all vertices $v_j \in F$ with $s - \Delta$ equal to zero are determined, where $s_{ui} = d(u_i, v_j) - w(u_i) - w(v_j)$ and $u_i \in S$. These edges can be determined from the priority queues. For each edge $(u_i, v_j)$ in this category, the vertex $v_j$ is added to an alternating tree and deleted from $F$.

In the algorithm of Vaidya, each time a vertex is deleted from a subset $F_i$, the WVD for $F_i$ is reconstructed and the edges shortest$[u_i, F_i]$, for all $u_i \in S_2$ are recomputed. Here, we delay the reconstruction till deletions are made from at least $\sqrt{n/p} + 1$ subsubsets $F_{tk}$ before reconstructing the WVD’s for the subsets from which vertices have been deleted.

### 4.2 Deleting Vertices from $F$

To delete the vertices $v_j$ from $F$, we determine subsubsets $F_{tk}$ to which the vertices $v_j$ belong. Let the set of these subsubsets be $\Phi$ and $|\Phi|$ be $n$. Let the total number of subsubsets from which vertices have been deleted be $\gamma$. If $\gamma > \sqrt{n/p}$, then after deleting the vertices $v_j$, the WVD's for $\sqrt{n/p} + 1$ of the $F_i$'s, together with their $F_{tk}$'s, are reconstructed in $O(\sqrt{n \log n} \times (\sqrt{n/p})/p) + O(n \log n/p^2)$ time. This process may be repeated $O(\sqrt{n \times p})$ times over a phase, resulting in $O(n \sqrt{n \log n/p})$ time.

If, however, $\gamma \leq \sqrt{n/p}$, we update shortest$[u_i, F_{tk}]$ for all vertices $u_i \in S_2$ and subsets $F_{tk}$ from which vertices have just been deleted as follows. We note that $\eta \times \sqrt{n}$ edges may have to be updated, using $\eta$ subsubsets $F_{tk}$ which may belong to at most $\eta$ subsets and $\sqrt{n}$ vertices in $S_2$. First we determine the shortest edge between a vertex $u_i$ in $S_2$ and a sub-subset $F_{tk}$ in $\Phi$. We compute the shortest edge using the edges $(u_i, v_j)$, $u_i \in S_2$, $v_j \in F_{tk}$. Each processor will determine $O(\sqrt{n/p})$ such edges. Each vertex in $S_2$ will have $\eta$ edges incident to it to be determined. We liken each vertex in $S_2$ to a set with $\eta$ elements (the edges). With this idea, we can proportionately assign sets of processors to each vertex $u_i$ in $S_2$ in $O(\sqrt{n/p} + \log p)$ time [12].

The distances can be determined in $O(\eta \sqrt{n/p} \times \sqrt{n/p})$ or $O(\eta n/p^2)$ time, where $\sqrt{n/p} = |F_{tk}|$. $F_{tk} \in \Phi$. Adding the previous two execution times, we get $O(\eta n/p^2 + \sqrt{n/p} + \log p)$ time. The sum of $\eta$ over a phase is at most $n$. This process may be repeated $n$ times, resulting in $O(n^2/p^2 + n \sqrt{n/p} + n \log p)$ time per phase.

To complete the update, we use the remaining $O(p)$ subsubsets per subset, we determine the edge shortest$[S_2, F_{tk}]$, using the already computed and available edges shortest$[S_2, F_{tk}]$, in $O(n \times p/p \times \log n + \log p)$ time per subset or $O(n \log n + n \log p)$ time in a phase.

### 4.3 Adding Vertices to $S_2$

We update the edge shortest$[S_2, F_{tk}]$, for all subsets $F_{tk}$ when vertices $u_i$ are added to $S_2$. Let $S_2 = S_2 \cup \{u_i\}$. The edge shortest$[S_2, F_{tk}] = \min\{\text{shortest}[S_2, F_{tk}], \text{shortest}[^{\{u_i\}} S_2, F_{tk}^]\}$

The vertices $u_i \in U$ such that an edge $(u_i, v_j)$ is in $M$, where $v_j$ is just deleted from $F$, are inserted into an alternating tree. Let the total number of these vertices $u_i$ be $\tau$. If the sum of $\tau$ and $|S_2|$ is less than $\sqrt{n}$, the $u_i$'s are included in $S_2$. For each $u_i$ included in $S_2$, $u_i$ is broadcast to all processors. The processors then determine the edges shortest$[u_i, F_{tk}]$ for all $F_{tk}$. In doing this, we first compute these edges for the subsets $F_{tk}$ from which vertices have been deleted before computing the edges for the sets $F_{tk}$ from which no vertex has been deleted.

We determine the edges shortest$[u_i, F_{tk}]$ for the $O(\sqrt{n/p})$ sets $F_{tk}$ from which vertices have been deleted, since vertices are deleted from $\sqrt{n/p}$ subsubsets, using the subsubsets of these subsets.

First we directly compute shortest$[u_i, F_{tk}]$ for all the subsubsets $F_{tk}$ for each subset $F_{tk}$ from which vertices have been deleted. In $O(\sqrt{n/p} + \log p)$ time, we can assign sets of processors proportionately to the subsets, with respect to the number of subsubsets from which vertices have been deleted. Then in $O(\sqrt{n/p} \times \sqrt{n/p}/p + \log p)$ or $O(n \log n + \log p)$ time the edges shortest$[u_i, F_{tk}]$ for all the subset $F_{tk}$ can be determined.

Secondly, we use the WVD's of the unaltered subsubsets $F_{tk}$ to compute the edges shortest$[u_i, F_{tk}]$ in $O(\sqrt{n/p} \times \sqrt{n/p} \times \log n + \log p)$ or $O(\sqrt{n \log n} + \log p)$ time. Using the edges shortest$[u_i, F_{tk}]$ obtained in the previous paragraph, and shortest$[u_i, F_{tk}]$, the edge shortest$[u_i, F_{tk}]$ can be determined in $O(\sqrt{n \log n/p + \log p})$ extra time.
The edges $\text{shortest}[u_t, F_t]$ are computed for the $F_t$'s unaltered in $O(\sqrt{n} \log n/p)$ time. Combining this time with the previous running times, we obtain $O(\sqrt{n} \log n/p + \log p)$ per update. The sum of $t$ in a phase is $n$. Therefore, the computation of the new edges $\text{shortest}[u_k, F_t]$ can be done in $O(n \sqrt{n} \log n/p + n \log p)$ in a phase.

However, if $|S_2| + \tau$ is greater than $\sqrt{n}$, then $S_2$ and $\sqrt{n} - |S_2|$ $u_k$'s are added to $S_1$. $S_2$ is added to a new subset $S_1'$ of $S_1$. A WVD for $S_1'$ is constructed and $\text{shortest}[v_j, S_1']$ is updated to the shortest of the edges $\text{shortest}[v_j, S_1]$ and $\text{shortest}[v_j, S_1']$. The priority queue $PQ_1$ is then updated appropriately in $O(\sqrt{n} \log n)$ time. In a phase there will be $\sqrt{n}$ such computations requiring $O(n \log n)$ time. The remaining $\tau - (\sqrt{n} - |S_2|)$ $u_k$'s are added to $S_2$. It is not hard to show that the priority queue $PQ_2$ can be maintained in the same manner as $PQ_1$, and updated in $O(n \sqrt{n} \log n/p + n \log n)$ time in a phase.

Summing all the time complexities obtained per phase, and simplifying we obtain $O(n^2/p + n \log n + n \log n/p)$ time per phase. Multiplying the total time by $n$ gives the time in Theorem 2. For $p \geq \sqrt{n}/\log n$ we achieve a perfect speedup algorithm.

5 Concluding Remarks

We have proposed a perfect speedup parallel algorithm for the assignment problem on the Euclidean plane. The algorithm requires $p < \sqrt{n}$ to work correctly, and $\sqrt{n}/\log n \leq p < \sqrt{n}$ and $\sqrt{n}/p$ is not a constant for a perfect speedup, where $p$ is the number of processors. Increasing the range of validity is an open problem. Further, the algorithm described here is for determining the minimum weight matching. Another open problem is to design a perfect speedup parallel algorithm for the maximum weight case. Also, an efficient parallel algorithm for the same problem that runs on an interconnection network with no shared memory is yet to be developed.

References


