Concurrent Implementation Of Munkres Algorithm

T. D. Gottschalk
California Institute of Technology
Pasadena, CA 91125

April 7, 1990

1 Introduction

The so-called *Assignment Problem* is of considerable importance in a variety of applications, and can be stated as follows. Let

$$\mathcal{A} \equiv \{a_1, a_2, \ldots, a_{N_A}\} \tag{1}$$

and

$$\mathcal{B} \equiv \{b_1, b_2, \ldots, b_{N_B}\} \tag{2}$$

be two sets of items and let

$$d_{ij} \equiv d[a_i, b_j] \geq 0,\ a_i \in \mathcal{A},\ b_j \in \mathcal{B} \tag{3}$$

be a measure of the distance (dissimilarity) between individual items from the two lists. Taking $N_A \leq N_B$, the objective of the assignment problem is to find the particular mapping

$$i \mapsto \Pi(i),\ 1 \leq i \leq N_A,\ 1 \leq \Pi(i) \leq N_B \tag{4}$$

$$i \neq j \Rightarrow \Pi(i) \neq \Pi(j) \tag{5}$$

such that the total association score

$$S_{TOT} \equiv \sum_{i=1}^{N_A} d[i, \Pi(i)] \tag{6}$$

is minimized over all permutations Π.

For $N_A \leq N_B$, the naive (exhaustive search) complexity of the assignment problem is $O[N_B!/(N_B - N_A)!]$. There are, however, a variety of exact solutions to the assignment problem with reduced complexity $O[N_A^2 N_B]$, (Refs.[1-3]). Section 2 briefly describes one such method, Munkres Algorithm [2], and presents a particular sequential implementation. Performance of the algorithm is examined for the particularly nasty problem of associating lists of random points within the unit square. In Section 3, the algorithm is generalized for concurrent execution, and performance results for runs on the MarkIII hypercube are presented.

2 The Sequential Algorithm

The input to the assignment problem is the matrix $D \equiv \{d_{ij}\}$ of dissimilarities from Eq.(3). The first point to note is that the particular assignment which minimizes Eq.(6) is not altered if a *fixed* value is added to or subtracted from all entries in any row or column of the cost matrix D. Exploiting this fact, Munkres solution to the Assignment Problem can be divided into two parts

M1 : Modifications of the distance matrix D by row/column subtractions, creating a (large) number of zero enties.

M2 : With $\{R_Z(i)\}$ denoting the row indices of all zeros in column i, construction of a so-called *Minimal Representative Set*, meaning a distinct selection $R_Z(i)$ for each i, such that $i \neq j \Rightarrow R_Z(i) \neq R_Z(j)$.

The steps of Munkres algorithm generally follow those in the constructive proof of P. Hall's theorem on Minimal Representative Sets.

The preceding paragraph provides a hopelessly incomplete hint as to the number theoretic basis for Munkres Algorithm. The particular implementation of Munkres algorithm used in this work is as described in Chapter 14 of Ref.[3]. To be definite, take $N_A \leq N_B$, and let the columns of the distance matrix be associated with items from list $\mathcal{A}$. The first step is to subtract the smallest item in each column from all entries in the column. The rest of the algorithm can be viewed as a search for *special* zero entries (starred zeros Z^*), and proceeds as follows:

Munkres Algorithm

Step 1 : Setup

1. Find a zero Z in the distance matrix.
2. If there is no starred zero already in its row or column, star this zero.

0-8186-2113-3/90/0000/0052$01.00 © 1990 IEEE

3. Repeat steps 1.1, 1.2 until all zeros have been considered.

Step 2 : Z^* Count, Solution Assessment.

1. Cover every column containing a Z^*.
2. Terminate the algorithm if all columns are covered. In this case, the locations of the Z^* entries in the matrix provide the solution to the assignment problem.

Step 3 : Main Zero Search

1. Find an uncovered Z in the distance matrix and prime it, $Z \mapsto Z'$. If no such zero exists, go to Step 5
2. If No Z^* exists in the row of the Z', go to Step 4.
3. If a Z^* exists, cover this row and uncover the column of the Z^*. Return to Step 3.1 to find a new Z.

Step 4 : Increment Set Of Starred Zeros

1. Construct the 'Alternating Sequence' of primed and starred zeros:

   Z_0 : Unpaired Z' from Step 3.2.

   Z_1 : The Z^* in the column of Z_0

   Z_{2N} : The Z' in the row of Z_{2N-1}, *if* such a zero exists.

   Z_{2N+1} : The Z^* in the column of Z_{2N}.

   the sequence eventually terminates with an unpaired $Z' = Z_{2N}$ for some N.
2. Unstar each starred zero of the sequence.
3. Star each primed zero of the sequence, thus increasing the number of starred zeros by one.
4. Erase all primes, uncover all columns and rows, and return to Step 2.

Step 5 : New Zero Manufactures

1. Let h be the smallest uncovered entry in the (modified) distance matrix.
2. Add h to all covered rows.
3. Subtract h from all uncovered columns
4. Return to Step 3, without altering stars, primes or covers.

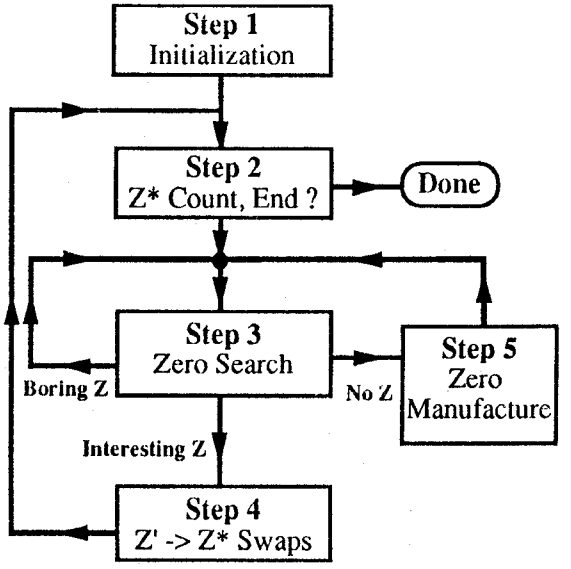

Figure 1: Flowchart for Munkres algorithm

A (very) schematic flowchart for the algorithm is shown in Fig.(1). Note that Steps 1,5 of the algorithm overwrite the original distance matrix.

The preceeding algorithm involves flags (starred or primed) associated with zero entries in the distance matrix, as well as 'Covered' tags associated with individual rows and columns. The implementation of the zero tagging is done by first noting that there is *at most* one Z^* or Z' in any row or column. The covers and zero tags of the algorithm are accordingly implemented using five simple arrays:

CC(k) : Covered column tags, $1 \leq k \leq N_{COLS}$.

CR(j) : Covered row tags, $1 \leq j \leq N_{ROWS}$

ZS(k) : Z^* locators for columns of the matrix. If positive, ZS(k) is the row index of the Z^* in the k^{th} column of the matrix.

ZR(j) : Z^* locators for rows of the matrix. If positive, ZR(j) is the column of the Z^* in the j^{th} row of the matrix.

ZP(j) : Z' locators for rows of the matrix. If positive, ZP(j) is the column of the Z' in the j^{th} row of the matrix.

Entries in the cover arrays CC and CR are one if the row or column is covered zero otherwise. Entries in the zero-locator arrays ZS, ZR and ZP are zero if no zero of the appropriate type exists in the indexed row or column.

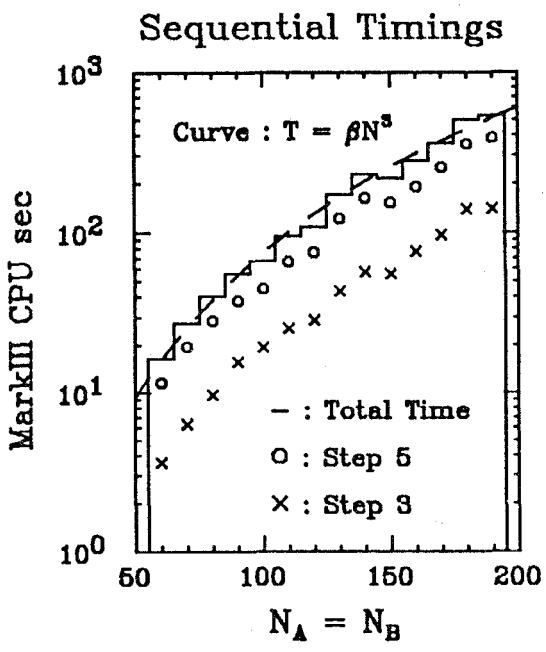

Figure 2: Timing results for the sequential algorithm versus problem size

With the Star-Prime-Cover scheme of the preceeding paragraph, a sequential implementation of Munkres algorithm is completely straightforward. At the beginning of Step 1, all cover and locator flags are set to zero, and the initial zero search provides an initial set of non-zero entries in ZS(). Step 2 sets appropriate entries in CC() to one and simply counts the covered columns. Steps 3 and 5 are trivially implemented in terms of the Cover/Zero arrays and the 'Alternating Sequence' for Step 4 is readily constructed from the contents of ZS(), ZR() and ZP().

As an initial exploration of Munkres algorithm, consider the task of associating two lists of random points within a 2D unit square, taking the cost function in Eq.(3) to be the usual Cartesian distance. Figure(2) plots total CPU times for execution of Munkres algorithm for equal size lists versus list size. The vertical axis gives CPU times in seconds for one node of the MarkIII hypercube. The circles and crosses show the time spent in Steps 5 and 3, respectively. These two steps (zero search and zero manufacture) account for essentially *all* of the CPU time. For the 190×190 case, the total CPU time spent in Step 2 was about 0.9 CPU sec, and that spent in Step 4 was too small to be reliably measured. The large amounts of time spent in Steps 3 and 5 arise from the very large numbers of times these parts of the algorithm are executed. The 190×190 case involves 6109 entries into Step 3 and 593 entries into Step 5.

Since the zero searching in Step 3 of the algorithm is required so often, the implementation of this step is done with some care. The search for zeros is done column-by-column, and the code maintains pointers to both the last column searched and the most recently uncovered column (Step 3.3) in order to reduce the time spent on subsequent re-entries to the Step 3 box of Fig.(1).

The dashed line if Fig.(2) indicates the nominal $\Delta T \propto N^3$ scaling predicted for Munkres algorithm. By and large, the timing results in Fig.(2) are consistent with this expected behavior. It should be noted, however, that both the nature of this scaling and the coefficient of N^3 are very dependent on the nature of the data sets. Consider, for example, two identical trivial lists

$$a_i \equiv b_i \equiv i, \; 1 \leq i \leq N \tag{7}$$

with the distance between items given by the absolute value function. For the data sets in Eq.(7), the preliminaries and Step 1 of Munkres algorithm completely solve the association in a time which scales as N^2 . In contrast, the random point association problem is a much greater challenge for the algorithm, as nominal pairings indicated by the initial nearest-neighbor searches of the preliminary step are tediously undone in the creation of the staircase-like sequence of zeros needed for Step 4. As a brief, instructive illustration of nature of this processing, Fig.(3) plots the CPU time *Per Step* for the last passes through the outer loop of Fig.(1) for the 150×150 assignment problem (recall that each pass through the outer loop increases the Z^* count by one). The processing load per step is seen to be highly non-uniform.

3 The Concurrent Algorithm

The timing results from Fig.(2) clearly dictate the manner in which the calculations in Munkres algorithm should be distributes among the nodes of a hypercube for concurrent execution. The zero and minimum element searches for Steps 3 and 5 are the most time consuming and should be done concurrently. In contrast, the essentially bookkeeping tasks associated with Steps 2 and 4 require insignificant CPU time and are most naturally done in lockstep (i.e., all nodes of the hypercube perform the same calculations on the same data at the same time). The details of the concurrent algorithm are as follows.

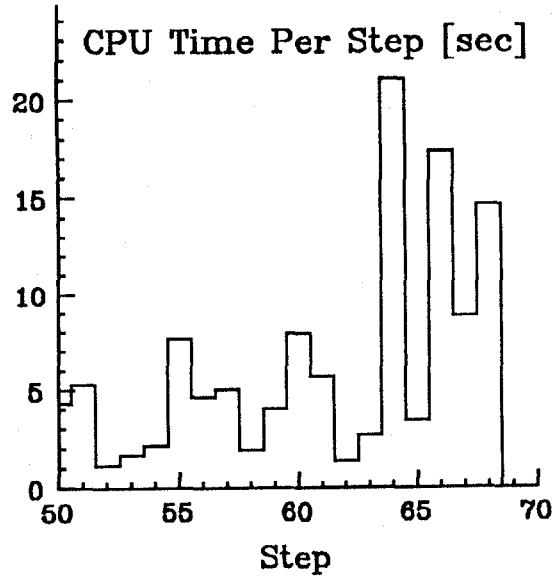

Figure 3: Times per loop (i.e., $N[Z^*]$ increment) for the last several loops in the solution of the 150×150 problem.

Data Decomposition

The distance matrix $\{d_{ij}\}$ is distributed across the nodes of the hypercube, with entire columns assigned to individual nodes. (This assumes, effectively, that $N_{COLS} \gg N_{NODES}$, which is always the case for assignment problems which are big enough to be 'interesting'.) The cover and zero locator lists defined in Section 2 are duplicated on all nodes.

Task Decomposition

The concurrent implementation of Step 5 is particularly trivial. Each node first finds its own minimum uncovered value, setting this value to some 'infinite' token if all columns assigned to the node are covered. A simple loop on communication channels determines the global minimum among the node-by-node minimum values, and each node then modifies the contents of its local portion of the distance matrix according to Steps(5.2,5.3).

The concurrent implementation of Step 3 is just slightly more awkward. On entry to Step 3, each node searches for zeros according to the rules of Section 2, and fills a 3-element status list:

$$L[j] \equiv L[\text{Node}_j] \equiv \{S, k_{ROW}, k_{COL}\} \qquad (8)$$

where S is a zero-search status flag,

$$S \equiv \begin{cases} -1 & \text{No } Z \text{ was found} \\ 0 & Z \text{ with } Z^* \text{ in row (Boring)} \\ 1 & Z \text{ without } Z^* \text{ (Interesting)} \end{cases} \qquad (9)$$

If the status is non-negative, the last two entries in the status list specify the location of the found zero. A simple channel loop is used to collect the individual status lists of each node into all nodes, and the action taken next by the program is as follows:

- If all nodes give negative status (no Z found), all nodes proceed to Step 5.
- If any node gives status 1, all nodes proceed to Step 4 for lockstep updates of the zero location lists, using the row-column indices of the node which gave status 1 as the starting point for Step 4.1. If more than one node returns status 1 (highly unlikely, in practice), only the first such node (lower node number) is used.
- If all zeros uncovered are 'Boring', the cover-switching in Step 3.3 of the algorithm is performed. This is done in lockstep, processing the Z's returned by the nodes in order of increasing node number. Note that the cover rearrangements performed for one node may well cover a Z returned by a node with higher node number. In such cases, the nominal Z returned by the later node is simply ignored.

It is worth emphasizing that only the actual *searches* for zero and minimum entries in Steps 3 and 5 are done concurrently. The updates of the cover and zero locator lists are done in unison.

The concurrent algorithm has been implemented on the MarkIII hypercube, and has been tested against random point association tasks for a variety of list sizes. Before examining results of these tests, however, it is worth noting that the concurrent implementation is not particularly dependent on the hypercube topology. The only communication-dependent parts of the algorithm are

1. Determination of the ensemble-wide minimum value for Step 5.
2. Collection of the local Step 3 status lists (Eq.(9).

either of which could be easily done for almost any MIMD architecture.

Table 1 presents performance results for the association of random lists of 200 points on the MarkIII

N[Nodes]	1	2	4	8
T[Total]	654.83	372.70	205.48	119.25
T[Step 3]	183.80	128.04	81.59	56.66
T[Step 5]	462.06	237.54	117.39	57.94
ϵTotal	-	0.878	0.800	0.686
ϵStep 3	-	0.718	0.563	0.405
ϵStep 5	-	0.973	0.984	0.997
N[Step 3]	7075	4837	3483	2778

Table 1: Concurrent performance For 200×200 random points

N[Nodes]	1	2	4	8
T[Total]	68.08	38.79	23.11	16.40
T[Step 3]	19.63	13.09	9.69	8.00
T[Step 5]	44.99	22.99	11.79	6.16
ϵTotal	-	0.878	0.736	0.519
ϵStep 3	-	0.750	0.506	0.307
ϵStep 5	-	0.978	0.954	0.913
N[Step 3]	2029	1430	1134	991

Table 2: Concurrent performance For 100×100 random points

hypercube for various cube dimensions. (For consistency, of course, the same input lists are used for all runs.) Time values are given in CPU seconds for the total execution time, as well as the time spent in Steps 3 and 5. Also given are the standard concurrent execution efficiencies,

$$\epsilon_N \equiv \frac{T[\text{ 1 Node }]}{N \times T[\text{ } N \text{ Nodes }]} \tag{10}$$

as well as the numbers of times the Step 3 box of Fig.(1) is entered during execution of the algorithm. The numbers of entries into the other boxes of Fig.(1) are independent of the hypercube dimension.

There is an aspect of the timing results in Table 1 which should be noted. Namely, essentially *all* inefficiencies of the concurrent algorithm are associated with Step 3 for 2 Nodes compared to Step 3 for 1 Node. The times spent in Step 5 are approximately halved for each increase in the dimension of the hypercube. However, the efficiencies associated with the zero searching in Step 3 are rather poorer, particularly for larger numbers of nodes.

At a simple, qualitative level, the inefficiencies associated with Step 3 are readily understood. Consider the task of finding a single zero located somewhere inside an $N \times N$ matrix. The mean sequential search time is

$$\langle T_{\text{Search}}[\text{ 1 Node }]\rangle \propto (N \times N)/2 \tag{11}$$

since, on average, half of the entries of the matrix will be examined before the zero is found. Now consider the same zero search on two nodes. The node which has the half of the matrix containing the zero will find it in about half the time of Eq.(11). *However*, the other node will *always* search through all of its $N \times N/2$ items before returning a null status for Eq.(9). Since the node which found the zero must wait for the other node before the (lockstep) modifications of zero locators and cover tags, the node without the zero determines the actual time spent in Step 3, so that

$$\langle T_{\text{Search}}[\text{ 2 Nodes }]\rangle \approx \langle T_{\text{Search}}[\text{ 1 Node }]\rangle \tag{12}$$

In the full program, the concurrent bottleneck is not as bad as Eq.(12) would imply. As noted above, the concurrent algorithm can process multiple 'Boring' Z's in a single pass through Step 3. The frequency of such multiple Z's per step can be estimated by noting the decreasing number of times Step 3 is entered with increasing hypercube dimension, as indicated in Table 1. Moreover, each node maintains a counter of the last column searched during Step 3. On subsequent re-entries, columns prior to this marked column are searched for zeros only if they have had their cover tag changed during the prior Step 3 processing. While each of these algorithm elements does diminish the problems associated with Eq.(12), the fact remains that the search for zero entries in the distributed distance matrix is the least efficient step in concurrent implementations of Munkres algorithm.

The results presented in Table 1 demonstrate that an efficient implementation of Munkres algorithm is certainly feasible. It is next interesting to examine how these efficiencies change as the problem size is varied.

The results shown in Tables 2,3 demonstrate an improvement of concurrent efficiencies with increasing problem size - the expected result. For the 100×100 problem on 8 nodes, the efficiency is only about 50problem is too small for 8 nodes, with only 12 or 13 columns of the distance matrix assigned to individual nodes.

While the performance results in Tables 1-3 are certainly acceptable, it is nonetheless interesting to investigate possible improvements of efficiency for the zero searches in Step 3. The obvious candidate for an algorithm modification is some sort of checkpoint-

N[Nodes]	1	2	4	8
T[Total]	2046.91	1154.27	622.53	353.30
T[Step 3]	585.61	399.41	235.49	154.57
T[Step 5]	1442.22	742.90	377.89	188.59
ϵTotal	-	0.887	0.822	0.728
ϵStep 3	-	0.733	0.621	0.473
ϵStep 5	-	0.971	0.954	0.956
N[Step 3]	13250	8583	5785	4365

Table 3: Concurrent performance For 300×300 random points

ing : at intermediate times during the zero search, the nodes exchange a 'Zero Found Yet ?' status flag, with all nodes breaking out of the zero search loop if any node returns a positive result.

For message passing machines such as the MarkIII, the checkpointing scheme is of little value, as the time spent in individual entries to Step 3 are not enormous compared to the node-to-node communication time. For example, for the 2-node solution of the 300×300 problem, the mean time for a single entry to Step 3 is only about 46 msec, compared to a typical node-to-node communications time which can be a significant fraction of a millisecond. The time required to perform a single Step 3 calculation is not large compared to node-to-node communications. As a (not unexpected) consequence, all attempts to improve the Step 3 efficiencies through various 'Found Anything ?' schemes were completely unsuccessful.

The checkpointing difficulties for a message-passing machine could disappear, of course, on a shared memory machine. If the zero-search status flags for the various nodes could be kept in memory locations readily (i.e., rapidly) accessible to all nodes, the problems of the preceding paragraph might be eliminated. It would be interesting to determine whether significant improvements on the (already good) efficiencies of the concurrent Munkres algorithm could be achieved on a shared memory machine.

References

1. F. Burgeios and J. C. Lassalle, 'An Extension of Munkres Algorithm for the Assignment Problem to Rectangular Matrices', *Comm. of the ACM*, 14(1971)802.

2. H. W. Kuhn, 'The Hungarian Method for the Assignment Problem', *Naval Research Logistics Quarterly*, 2(1955)83.

3. S. S. Blackman, *Multiple-Target Tracking with Radar Applications*, Dedham, MA: Artech House(1986).