Parallel Computing 11 (1989) 151–162
North-Holland

Algorithms for assignment problems on an array processor

A.M. FRIEZE
Department of Mathematics, Carnegie-Mellon University, Pittsburgh, U.S.A.

J. YADEGAR *
DAP Support Unit, Queen Mary College, London University, United Kingdom

S. EL-HORBATY
Department of Mathematics, Ain Shams University, Cairo, Egypt

D. PARKINSON
DAP Support Unit, Queen Mary College, London University, United Kingdom, and Active Memory Technology Ltd., Reading, United Kingdom

Received August 1987

Abstract. The innovation of parallel computers has added a new dimension to the design of algorithms. Parallel programming is not a simple extension of serial programming. We describe parallel algorithms for the quadratic assignment problem and present our computational experience using the massively parallel processor, DAP. We further report the speedup obtained by parallelising algorithms for solving the 2-dimensional and 3-dimensional assignment problems on the DAP.

Keywords. Quadratic assignment problem, 2- and 3-dimensional assignment problems, heuristic parallel algorithm, distributed array processor, SIMD computer.

1. Introduction

The problem of finding an optimal assignment arises under many circumstances: for example, assigning indivisible resources to different geographical locations, assigning students to schools who need to be supervised by tutors, or simply assigning men to jobs. These are examples of *combinatorial optimisation problems*, in which it is required to determine an optimal solution from a large but finite set of feasible solutions.

The major difficulty with most of the problems of this type is in finding efficient algorithms to solve them. Indeed, many of these problems are known to belong to the special class of NP-hard problems, see [10].

* Currently at Active Memory Technology, Inc. Irvine, CA 92714, U.S.A.

The innovation of parallel computers has added a new dimension to the design of algorithms. *Parallel programming* is not a simple extension of serial programming. Our motivation in this research has been to develop parallel algorithms for such problems, in particular the well-known quadratic assignment problem, the 2-dimensional and 3-dimensional assignment problems. These algorithms have then been implemented on the Distributed Array Processor (DAP).

The general principle of the DAP is that of a SIMD (Single Instruction stream-Multiple Data stream) machine as defined by Flynn [6]. On an $n \times n$ DAP, one can perform up to n^2 operations (of the same type) in parallel, that is, simultaneously. This parallel processing capability of the DAP is achieved by an $n \times n$ matrix of processors, called Processing Elements, each of which may operate independently on its own local store. Thus, it is convenient to think of the DAP as a square array of processors placed on a 2-dimensional grid in which each processor can communicate directly with its four neighbours. Extra communication facilities, along rows and columns, coupled with the bit serial nature of the processors allow the DAP to exhibit many properties of the associative or content addressable processors [8].

For further details on the DAP and its programming language, DAP-Fortran, we refer the reader to the papers of Gostick [11], Parkinson [19,20], Reddaway [22], and the book by Hockney and Jesshope [13].

The rest of the paper is organised as follows: In Section 2, after defining the quadratic assignment problem, we describe two heuristic parallel algorithms to solve this problem and present our computational results in Section 2.4. We outline very briefly in Section 3 a parallel version of the primal dual method for solving the assignment problem, and give computational results in Section 3.1. In Section 4, we describe the 3-dimensional assignment problem and discuss a parallel heuristic for its solution followed by some results in Section 4.1. We conclude the paper in Section 5.

2. The Quadratic Assignment Problem – QAP

This is the problem of assigning a number of discrete facilities to a number of discrete locations when there is an interchange or material flow between each pair of facilities. Unlike the classical assignment problem, the cost of assigning each facility is dependent on the assignment of the rest of the facilities.

The QAP has proved to be extremely difficult to solve optimally in practice as well as in theory. In its most general form it can be succinctly defined as follows:

$$\text{minimise } f(\phi) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{i\phi(i)j\phi(j)}, \tag{2.1}$$

$$\text{subject to } \phi \in S_n$$

where S_n is the set of permutations of $[n] = \{1, 2, \ldots, n\}$ and $a_{ipjq} \geqslant 0$, for $i, p, j, q \in [n]$, is the cost of pair assignment of facilities i and j to locations p and q.

The QAP models a number of practical problems – see [4] for a recent survey – but the problem becomes very difficult for n as low as 20 and 'impossible' for $n = 30$.

There has therefore been intensive research to find good heuristics for this problem. While it is NP-hard to approximate the optimal value within a constant factor, see [23], probabilistic analysis, [3], suggests that it is usually not too hard to find a good solution. (Dyer et al. [5] show however that it is usually very hard to find an exact solution by branch and bound.)

2.1. *r-Optimality*

A standard procedure, which can be used as an add-on to any heuristic is that of checking for r-optimality, provided that r is reasonably small, $r=2$ or 3 say.

A permutation ϕ is said to be r-optimal if

$$f(\phi) \leqslant f(\psi) \quad \text{for all } \psi \in N_r(\phi) \tag{2.2}$$

where

$$N_r(\phi) = \{\psi \in S_n:\ |\{i:\ \psi(i) \neq \phi(i)\}| \leqslant r\}$$
$$(=\text{the set of } r \text{ neighbours of } \phi).$$

Since $|N_r(\phi)| = O(n^r)$ and it takes $O(n^2)$ time to evaluate f, one can check (2.2) and find an improved solution, if one exists, in $O(n^{r+2})$ time. This process can be repeated until an r-optimal solution is found. Even for $r=2$ and $n=30$, each test of (2.2) can be rather expensive.

Fortunately, some improvements are possible. In most cases that occur in practice the values a_{ipjq}'decompose' so that we can write

$$a_{ipjq} = \tfrac{1}{2} c_{ij} d_{pq} \quad \text{for } i,\ p,\ j,\ q \in [n]. \tag{2.3}$$

This is the so-called Koopmans–Beckmann [15] version of the problem. We will assume for simplicity that the terms $c_{ii} = d_{ii} = 0$ for $i \in [n]$ and that the matrices $\| c_{ij} \|$, $\| d_{ij} \|$ are both symmetric. (The substitution of $c'_{ij} = (c_{ij} + c_{ji})/2$ and $c''_{ij} = (c_{ij} - c_{ji})/2$ and similarly for d_{ij} produces the sum of two symmetric functions. The values c_{ii}, d_{ii} represent 'linear' terms and it is easy to modify what follows to accomodate them.)

Thus f as defined In (2.1) now becomes

$$f(\phi) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{ij} d_{\phi(i)\phi(j)}. \tag{2.4}$$

Los [18] showed that it is possible to check for the satisfaction of (2.2) much faster than is immediately obvious. In this paper we show how a modification of Los's idea can be used in conjunction with parallel computation in order to substantially reduce the running time of a basic algorithm for finding r-optimal solutions for $r=2$ or 3.

2.2. *A 2-optimising heuristic*

Let us first consider the case $r=2$. For $\phi \in S_n$ and distinct $k,\ l \in [n]$, let ϕ^{kl} be the permutation obtained by interchanging the images of k, l; that is,

$$\phi^{kl}(i) = \begin{cases} \phi(i) & \text{if } i \neq k,\ l, \\ \phi(l) & \text{if } i = k, \\ \phi(k) & \text{if } i = 1. \end{cases}$$

Now let

$$\text{DIFF}(\phi;\ k,\ l) = f(\phi) - f(\phi^{kl}) \quad \text{for } k,\ l \in [n].$$

Clearly ϕ is 2-optimal if and only if $\text{DIFF}(\phi;\ k,\ l) \leqslant 0$ for all $k,\ l \in [n]$. (Note that $\text{DIFF}(\phi;\ k,\ l) = \text{DIFF}(\phi;\ l,\ k)$ and $\text{DIFF}(\phi;\ k,\ k) = 0$.)

2-Optimality Algorithm

```
begin
  compute an initial permutation $\phi$; {see [4] for a review of good starting methods}
  2OPTIMAL := false; compute DIFF($\phi$,·,·);
  repeat
    if DIFF($\phi$;·,·) $\leqslant 0$ then 2OPTIMAL := true
    else begin
      let DIFF($\phi$; $k$, $l$) = max{DIFF($\phi$; $i$, $j$): $i$, $j \in [n]$};
      $\phi := \phi^{kl}$;
      re-compute (update) DIFF($\phi$;·,·)
    end
  until 2OPTIMAL
end
```

As suggested by the structure of the algorithm, the statement re-compute DIFF is executed differently from the statement compute DIFF and it is here that we can make some savings.

First note that

$$\mathrm{DIFF}(\phi;\ i,\ j) = A'(\phi;\ i,\ j) + A'(\phi;\ j,\ i) \quad \text{for } i,\ j \in [n]$$

where

$$A'(\phi;\ i,\ j) = \sum_{\substack{p=1 \\ \neq i,j}}^{n} c_{ip}\left(d_{\phi(i)\phi(p)} - d_{\phi(j)\phi(p)}\right) \tag{2.5}$$

is the change in cost attributed to i being reassigned to $\phi(j)$ from $\phi(i)$.

Now, by completing the sum in (2.5) from 1 to n, we can rewrite DIFF(ϕ; i, j) as

$$\mathrm{DIFF}(\phi;\ i,\ j) = A(\phi;\ i,\ j) + A(\phi;\ j,\ i) - 2c_{ij}d_{\phi(i)\phi(j)}$$

where

$$A(\phi;\ i,\ j) = \sum_{p=1}^{n} c_{ip}\left(d_{\phi(i)\phi(p)} - d_{\phi(j)\phi(p)}\right) \tag{2.6}$$

and $2c_{ij}d_{\phi(i)\phi(j)}$ is a 'correction' term.

Suppose then that at some stage we have DIFF, ϕ, k, l and we wish to re-compute DIFF. That is to calculate

$$\mathrm{DIFF}(\phi^{kl};\ i,\ j) = A(\phi^{kl};\ i,\ j) + A(\phi^{kl};\ j,\ i) - 2c_{ij}d_{\phi^{kl}(i)\phi^{kl}(j)} \quad \text{for } i,\ j \in [n].$$

Thus, for $i,\ j \in [n] - \{k,\ l\}$, let

$$\delta A(i,\ j) = A(\phi^{kl};\ i,\ j) - A(\phi;\ i,\ j).$$

Then it is not difficult to drive the following expression for $\delta A(i,\ j)$:

$$\delta A(i,\ j) = (c_{ik} - c_{il})\left(d_{\phi(i)\phi(l)} - d_{\phi(i)\phi(k)} + d_{\phi(j)\phi(k)} - d_{\phi(j)\phi(l)}\right). \tag{2.7}$$

To re-compute the kth and lth rows and columns of DIFF, we use (2.6).

2.2.1. *Computational considerations*

Consider first a normal serial computer. The initial computation of DIFF using (2.6) is $O(n^3)$ time (= number of arithmetic operations). However, to re-compute DIFF we can use (2.7) to compute DIFF(ϕ^{kl}; i, j) for $i,\ j \in [n] - \{k,\ l\}$ and (2.6) to compute the remaining entries, which results in an $O(n^2)$ time computation.

Table 1
Ratio of the DAP computation time for an operation compared to that of a 32-bit floating point matrix (element by element) multiplication

Function	$\frac{\text{Operation time}}{\text{Multiply time}}$
*	1.0
+	0.64
/	1.42
$A(i,) = V$ or $A(,j) = V$	0.15
MATR(V) or MATC(V)	0.11
SUMR(A) or SUMC(A)	1.28
MAXV(A)	0.21
TRAN(A)	2.60

The main purpose of this paper however is to present the results obtained when implementing the above approach on the Distributed Array Processor (DAP).

The abstract model of the computer that we use is that of a 2-dimensional mesh connected system with some extra communication facilities and computing power. Specifically, we have an array of n^2 processors P_{ij}, i, $j \in [n]$, where P_{ij} can communicate directly with its four neighbours, $P_{i-1,j}$, $P_{i+1,j}$, $P_{i,j-1}$, $P_{i,j+1}$, where $1-1 \equiv n$ and $n+1 \equiv 1$. In addition, processors are connected via row and column highways to a set of edge registers such that in unit time data can be selected from any set of processors, one per row (or column), into the corresponding register; or data can be broadcast from a register to some or all processors in the same row (or column). The DAP satisfies this abstract model.

We measure the complexity of the algorithm in terms of number of 'DAP operations'. Each one is executable very 'efficiently' on the DAP. These are to

(i) Compute the result matrix $\| a_{ij} \circ b_{ij} \|$ where $\circ = +, -, *$ or $/$. The $n \times n$ matrices $\| a_{ij} \|$ and $\| b_{ij} \|$ are stored one element per processor.

(ii) Overwrite a row or column of an $n \times n$ matrix with a given n-vector.

(iii) Given an n-vector V, construct an $n \times n$ matrix MATR(V) where each row is identical to V. Similarly, MATC(V) has each column identical to V.

(iv) Given an $n \times n$ matrix A, create an n-vector SUMR(A) in which component i is the sum of the elements of column i of A. Similarly, the SUMC(A) has its ith component equal to the sum of the elements in row i of A.

(v) Compute the maximum element of an $n \times n$ matrix.

Thus, using the DAP one is able to execute these operations in O(1) time. To get a feeling of the actual computational times for these operations on the existing DAPs with $n = 32$ and 64; Table 1 gives the ratio of the time for an operation compared to that for a 32-bit floating point matrix (element by element) multiplication as defined in (i) above. For $n = 32$ and $n = 64$, the times for matrix (element by element) multiplication are 210 and 270 μs respectively.

In Table 1, A is an $n \times n$ matrix and V is an n-vector. The assignment $A(i,) = V$ ($A(,j) = V$) overwrites the ith row (jth column) of A by V. Functions MATR, MATC, SUMR and SUMC have already been defined in (iii) and (iv). Function MAXV(A) returns a value equal to the maximum value in the matrix argument A, and the function TRAN(A) returns the transpose of A. For more detail on these functions, we refer the reader to [20,21].

In this model our algorithm requires

(a) O(n) DAP operations to compute DIFF 'from scratch' using (2.6),

(b) O(1) DAP operations to re-compute DIFF using (2.7).

It is assumed that at the beginning of each iteration

$$\text{Processor } P_{ij} \text{ contains the values } c_{ij} \text{ and } d_{\phi(i)\phi(j)} \text{ in its local memory.} \tag{2.8}$$

To achieve (2.8), there is an initial complete shuffling of the rows and columns of $\|d_{ij}\|$ using the permutation ϕ. Subsequently, once k and l are determined, as described in the 2-optimality algorithm, it is only necessary to interchange the kth row and column with the lth row and column of $\|d_{ij}\|$. This is an O(1) time operation on the DAP.

Details of (a). We compute $\|A(\phi;\ i,\ j)\|$ as defined in (2.6) and then compute $\mathrm{DIFF}(\phi;\cdot,\cdot)$ using type (i) DAP operations. We note that in order for processor P_{ij} to contain $\|A(\phi;\ j,\ i)\|$, it is only necessary to transpose the matrix $\|A(\phi;\ i,\ j)\|$.

For a *fixed* p $(1 \leqslant p \leqslant n)$, we compute matrices $B^{(p)} = \mathrm{MATC}((c_{1p},\dots,c_{np})) = \|b_{ij}^{(p)}\|$ and $E^{(p)} = \mathrm{MATC}((d_{\phi(1)\phi(p)},\dots,d_{\phi(n)\phi(p)})) - \mathrm{MATR}((d_{\phi(1)\phi(p)},\dots,d_{\phi(n)\phi(p)})) = \|e_{ij}^{(p)}\|$. We can then using type (i) operations, compute $\|a_{ij}^{(p-1)} + (b_{ij}^{(p)} * e_{ij}^{(p)})\|$, where $a_{ij}^{(p-1)} = \sum_{i=1}^{p-1} c_{ip}(d_{\phi(i)\phi(p)} - d_{\phi(j)\phi(p)})$ and $a_{ij}^{(0)} = 0$ for $i,\ j \in [n]$. When p takes the value n, we obtain $A(\phi;\cdot,\cdot)$.

The DAP-Fortran program to achieve (a), assuming (2.8), is as follows:

	Remarks
A = 0	$\|a_{ij}\| := 0$ for all $i,\ j$
DO 10 p = 1, n	
B = MATC(C(,p))	$\|b_{ij}\| := \|c_{ip}\|$ for all $i,\ j$
E = MATC(D(,p)) − MATR(D(,p))	$\|e_{ij}\| := \|d_{\phi(i)\phi(p)} - d_{\phi(j)\phi(p)}\|$ for all $i,\ j$
A = A + B∗E	$\|a_{ij}\| := \|a_{ij} + b_{ij}e_{ij}\|$ for all $i,\ j$
10 CONTINUE	
DIFF = A + TRAN(A) − 2∗C∗D	$\|\mathrm{DIFF}_{ij}\| := \|a_{ij} + a_{ji} - 2c_{ij}d_{\phi(i)\phi(j)}\|$ for all $i,\ j$

In this and subsequent programs $C = \|c_{ij}\|$, $D = \|d_{\phi(i)\phi(j)}\|$ and $A = \|a_{ij}\| = \|A(\phi;\ i,\ j)\|$.

Details of (b). In practice we actually update DIFF instead of re-computing it, using the fact that

$$\mathrm{DIFF}(\phi^{kl};\ i,\ j) = \mathrm{DIFF}(\phi;\ i,\ j) + \delta A(i,\ j) + \delta A(j,\ i)$$
$$\text{for } i,\ j \in [n] - \{k,\ l\}.$$

Having already given a flavour of a DAP-Fortran program and defined various functions in the language, it is easier to describe steps involved in (b) in terms of a DAP program as given below

	Remarks
B1 = MATC(C(,k) − C(,l))	$\|b1_{ij}\| := \|c_{ik} - c_{il}\|$ for all $i,\ j$
B2 = MATR(C(,l) − C(,k))	$\|b2_{ij}\| := \|c_{jl} - c_{jk}\|$ for all $i,\ j$
E = MATC(D(,l) − D(,k)) + MATR(D(,k) − D(,l))	$\|e_{ij}\| := \|d_{\phi(i)\phi(l)} - \mathrm{d}_{\phi(i)\phi(k)} + d_{\phi(j)\phi(k)} - d_{\phi(j)\phi(l)}\|$ for all $i,\ j$
DIFF = DIFF + (B1 + B2)∗E	$\|\mathrm{DIFF}_{ij}\| := \|\mathrm{DIFF}_{ij} + (b1_{ij} + b2_{ij})e_{ij}\|$ for all $i,\ j$

The element by element matrix multiplications $B1 * E$ and $B2 * E$ coincide with matrices δA and δA^{T} ($\mathrm{T} \equiv$ transpose) respectively, except for rows and columns $k,\ l$.

It remains only to compute the kth and lth rows and columns of DIFF. To do this, as stated earlier, we use (2.6). It is sufficient to show how to compute the kth (say) row and column of $A(\phi;\cdot,\cdot)$. This is done as follows:

	Remarks
C1 = MATR(C(k,))	$\|cl_{ij}\| := \|c_{kj}\|$ for all i, j
D1 = MATR(D(k,))	$\|dl_{ij}\| := \|d_{\phi(k)\phi(j)}\|$ for all i, j
A(k,) = SUMC(−C1 * (D − D1))	$a_{ki} := \sum_{j=1}^{n} - cl_{ij}(d_{\phi(i)\phi(j)} - dl_{ij})$ for all i
A(,k) = SUMC(C * (D − D1))	$a_{ik} := \sum_{j=1}^{n} c_{ij}(d_{\phi(i)\phi(j)} - dl_{ij})$ for all i

It is now immediate to see that to update DIFF is indeed O(1) DAP operations.

A program for 2-optimising problems of size $n \leqslant 64$ has been produced for the DAP Subroutine Library.

2.3. A 3-optimising heuristic

For $\phi \in S_n$ and distinct $k, l, m \in [n]$ let ϕ^{klm} denote the following member of $N_3(\phi)$:

$$\phi^{klm}(i) = \begin{cases} \phi(i) & \text{if } i \notin \{k, l, m\}, \\ \phi(l) & \text{if } i = k, \\ \phi(m) & \text{if } i = l, \\ \phi(k) & \text{if } i = m. \end{cases}$$

We extend the use of DIFF so that DIFF $(\phi;\ k, l, m) = f(\phi) - f(\phi^{klm})$.

3-Optimality Algorithm

```
begin
   compute an initial permutation $\phi$;
   3OPTIMAL := false;
   repeat
      2-optimise $\phi$ using the algorithm of Section 2.2;
      if DIFF($\phi;\cdot,\cdot,\cdot$) $\leqslant 0$ then 3OPTIMAL := true
      else begin
         let DIFF($\phi;\ k, l, m$) = max{DIFF($\phi;\ u, v, w$): $u, v, w \in [n]$};
         $\phi := \phi^{klm}$
      end
   until 3OPTIMAL
end
```

We should first check that on termination ϕ is 3-optimal. But this is immediate from the fact that ϕ is 2-optimal and

$$N_3(\phi) = N_2(\phi) \cup \{\phi^{klm}:\ k, l, m \in [n] \text{ and } k, l, m \text{ distinct}\}.$$

To see how to implement the algorithm efficiently we note first that

$$\begin{aligned} &\mathrm{DIFF}(\phi;\ u, v, w) \\ &\quad = A'(\phi;\ u, v) + c_{uw}(d_{\phi(v)\phi(w)} - d_{\phi(v)\phi(u)}) + A'(\phi;\ v, w) \\ &\qquad + c_{vu}(d_{\phi(w)\phi(u)} - d_{\phi(w)\phi(v)}) \\ &\qquad + A'(\phi;\ w, u) + c_{wv}(d_{\phi(u)\phi(v)} - d_{\phi(u)\phi(w)}) \\ &\quad = A(\phi;\ u, v) + A(\phi;\ v, w) + A(\phi;\ w, u) - c_{uv}(d_{\phi(u)\phi(v)} + d_{\phi(v)\phi(w)} - d_{\phi(w)\phi(u)}) \\ &\qquad - c_{vw}(d_{\phi(v)\phi(w)} + d_{\phi(w)\phi(u)} - d_{\phi(u)\phi(v)}) - c_{wu}(d_{\phi(w)\phi(u)} + d_{\phi(u)\phi(v)} - d_{\phi(v)\phi(w)}). \end{aligned}$$

(This is, of course, not obvious, but neither is it difficult to check, only tedious.)

Table 2
Comparison between running times of 'serial' and 'parallel' versions of the 2-optimality and 3-optimality algorithms on randomly generated problems

Size of problem n	2-optimality algorithm			3-optimality algorithm
	Average execution time on 64×64 DAP (ms)	Average execution time on ICL 2980 (ms)	Speedup factor of the DAP to 2980	Average execution time on 64×64 DAP (ms)
10	23.69	18.10	0.76	61.23
20	45.48	91.82	2.02	160.65
30	85.15	394.83	4.64	214.15
40	107.70	896.36	8.32	394.36
50	147.38	1980.22	13.44	694.94
60	156.60	3053.77	19.51	878.40
64	182.80	4054.81	22.18	1056.04

Note that after 2-optimising ϕ in the 2-optimality algorithm, the (updated) matrix $A(\phi;\cdot,\cdot)$ is available to us. We can then, for a fixed u (say $u=k$), compute DIFF$(\phi;\ k,\cdot,\cdot)$ and then see if there exist $l,\ m\in[n]$ such that DIFF$(\phi;\ k,\ l,\ m)>0$ using O(1) DAP operations. Thus, one can check for 3-optimality in $O(n)$ DAP operations. ($O(n^3)$ for a serial implementation as opposed to $O(n^5)$ for a crude comparison of $f(\phi)$ and $f(\psi)$ for $\psi\in N_3(\phi)$.)

It remains only to 're-compute' DIFF$(\phi;\cdot,\cdot,\cdot)$ for which we must update $A(\phi;\cdot,\cdot)$. But $\phi^{klm}=(\phi^{kl})^{lm}$ and so A can be updated using O(1) DAP operations by applying (2.7) twice.

2.4. Computational results

We have carried out some computational experiments to try to evaluate the efficiency of the algorithms as described in Sections 2.2 and 2.3. We have not reported on the qualities of the solutions obtained, as these are well documented in [18].

Two sets of data were used for testing the algorithms

(i) A series of randomly generated problems. Three matrices $C=\|c_{ij}\|$, $D=\|d_{ij}\|$ and $F=\|f_{ij}\|$ were generated using a uniform distribution such that $1\leqslant c_{ij},\ d_{ij},\ f_{ij}\leqslant 100$. The value f_{ij} represents the 'linear' term which is the fixed cost of assigning facility i to location j. For each value of n $(10\leqslant n\leqslant 64)$ 10 problems were generated. These results are summarised in Table 2. As we had a working version of Los's code for the 2-optimising case, we were able to compare the computation time of the parallel algorithm to that of the serial algorithm.

Table 3
Comparison between running times of 'serial' and 'parallel' versions of the 2-optimality and 3-optimality algorithms on the Steinberg problem

	Value of the local optimum assignment	Execution time (ms)
Los's serial 2-optimality algorithm	4236.497	344.00 (on CDC CYBER 74)
Parallel 2-optimality algorithm	4236.497	53.00 (on 64×64 DAP)
Parallel 3-optimality algorithm	4223.033	382.23 (on 64×64 DAP)

(ii) The Steinberg problem [24] using as the starting solution the solution reported by Graves and Whinston [12] which has the value 4344.975. These results are given in Table 3.

3. The assignment problem

The classical linear (or 2-dimensional) assignment problem is defined as the following integer program:

$$\begin{aligned}
&\text{minimise} && \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}, && \\
&\text{subject to} && \sum_{j=1}^{n} x_{ij}=1, && i=1,2,\dots,n,\\
& && \sum_{i=1}^{n} x_{ij}=1, && j=1,2,\dots,n,\\
& && x_{ij}=0 \text{ or } 1, && i,\ j=1,2,\dots,n.
\end{aligned}$$

This is a standard problem in Operational Research which is well studied and a number of efficient algorithms have been proposed for its solution; e.g. [1,2,14,16]. All of these can be implemented to run in time $O(n^3)$ on a serial machine.

One of the best known methods is the Hungarian or Primal Dual Method. This method seems to have the most promise in a parallel setting and we programmed a version of it in DAP-Fortran. The algorithm works with a dual solution $(u, v)\in\mathbb{R}^{2n}$ and the main tasks carried out by the algorithm are to:

(i) Find a maximum matching in the bipartite graph with edge set

$$\{(i,\ j):\ c_{ij}=u_i+v_j\}.$$

(ii) Update the dual solution by computing

$$\min\{c_{ij}-u_i-v_j:\ (i,\ j)\in I\times J\}$$

for suitably defined sets I, J, see [17] for details.

On a serial machine (ii) is very time consuming, but the DAP can do the minimisation in O(1) very rapidly. To carry out (i), we use the standard labelling algorithm of Ford and Fulkerson [7]. However, because of the 0–1 nature of the problem, this can be done using logical matrices which are represented by 1 bit in the DAP and hence is very fast.

Our experience has been limited to problems of size up to 64×64 and a program for solving problems of this size has been produced for the DAP Subroutine Library. Larger problems can be solved by using standard partitioning techniques, but we have not had the time to implement these.

3.1. Computational results

The results of running our program on randomly generated test problems (with uniform distribution on the interval [1.0, 10.0]) are summarised in Table 4.

Table 4
Comparison between running times of 'serial' and 'parallel' versions of the primal-dual algorithm for solving linear assignment problems

Size of problem	Time on ICL 2980 (ms)	Time on 64×64 DAP (ms)	Speedup factor of the DAP to 2980
10	36.74	6.33	5.80
20	65.25	11.32	5.76
30	179.66	20.65	8.70
40	264.06	21.63	12.21
50	624.66	30.91	20.21
60	1816.02	51.45	35.30
64	1259.95	29.04	43.39

We can see that as the problem get larger, so do the savings in computation time. We anticipate that for problems of size greater than 64, the speedup factor of at least 44 will be maintained.

4. The 3-dimensional assignment problem

This is a well-known generalisation of the assignment problem described in Section 3. This problem arises when three sets of entities, e.g. (persons, jobs and times) or (students, projects and teachers) have to be assigned together in order to maximise some objective function, see [9]. Expressed mathematically this is to

$$\text{maximise} \quad \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} v_{ijk}x_{ijk}, \tag{4.1}$$

$$\begin{aligned} \text{subject to} \quad & \sum_{j=1}^{n}\sum_{k=1}^{n} x_{ijk} = 1, && i = 1, 2, \ldots, n, \\ & \sum_{i=1}^{n}\sum_{k=1}^{n} x_{ijk} = 1, && j = 1, 2, \ldots, n, \\ & \sum_{i=1}^{n}\sum_{j=1}^{n} x_{ijk} = 1, && k = 1, 2, \ldots, n, \\ & x_{ijk} = 0 \text{ or } 1, && i, j, k = 1, 2, \ldots, n. \end{aligned} \tag{4.2}$$

This problem is NP-Hard, see [10], and the most efficient method of solution is branch and bound using Lagrangean relaxation [9]. By allocating Lagrange multipliers, $u_1, u_2, \ldots, u_n$ to equations (4.2) and taking them up into the objective function (4.1), we obtain the following relaxation:

$$\text{maximise} \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}y_{ij}, \tag{4.3}$$

$$\begin{aligned} \text{subject to} \quad & \sum_{j=1}^{n} y_{ij} = 1, \quad i = 1, 2, \ldots, n, \\ & \sum_{i=1}^{n} y_{ij} = 1, \quad j = 1, 2, \ldots, n, \\ & y_{ij} = 0 \text{ or } 1, \quad i, j = 1, 2, \ldots, n \end{aligned}$$

where $c_{ij} = \max\{(v_{ijk} - u_k): k = 1, 2, \ldots, n\} = v_{ijk(i,j)} - u_{k(i,j)}$ and $y_{ij} = x_{ijk(i,j)}$.

Table 5
Comparison between running times of 'serial' and 'parallel' versions of the algorithm for solving 3-dimensional assignment problems

Size of problem	Time on ICL 2980 (s)	Time on 64×64 DAP (s)	Speedup factor of the DAP to 2980
30	6.70	0.75	8.93
50	36.50	1.38	26.45
60	52.15	2.03	25.69

Now (4.3) is an assignment problem and can be solved as in Section 3. The only other difficulty is the actual calculation of the c_{ij}'s. This is very time consuming when done serially; it requires 3 DO-loops (in fact, construction of the matrix $\| c_{ij} \|$ was a bottleneck that had to be overcome if the method was to be practicable). But the calculation is done very quickly on the DAP. Indeed, we can reduce the 3 DO-loops to only 1 DO-loop. The DAP-Fortran program to achieve this is as follows:

```
                          Remark
    C = V(,,1) - U(1)     $\| c_{ij} \| := \| v_{ij1} \| - \| u_1 \|$ for all $i, j$
    DO 10 k = 2, n
    T = V(,,k) - U(k)     $\| t_{ij} \| := \| v_{ijk} \| - \| u_k \|$ for all $i, j$
    C(C .LT. T) = T       $\| c_{ij} \| := \| t_{ij} \|$ whenever the former is less than the latter,
                            for all $i, j$
10  CONTINUE
```

In the expression $V(,,k) - U(k)$, the compiler first automatically expands $U(k)$ into a matrix each component of which has the value $U(k)$ and then performs the subtraction.

4.1. Computational results

The results of running our program on randomly generated test problems are summarised in Table 5. We have not reported on the quality of the solutions obtained, as these are well documented in [9].

5. Conclusion [1]

The idea of applying parallel programming techniques to solve combinatorial optimisation problems is promising and deserves much more attention. Parallel programming could save significant amount of computation times. For example, by parallising the primal dual algorithm on the DAP, we achieved speedup factor of nearly 44 times for solving assignment problems of size 64. Similar savings, though not as much, were also obtained for the 3-dimensional assignment problem and the QAP. Contrary to what Los concluded in his paper [18], we were able, using the DAP, to apply the 3-optimality algorithm in acceptable times to improve solutions to QAPs of size > 24.

[1] The computations reported in this work were performed on a 64×64 DAP system designed by ICL. All software and hardware rights to the original machine were sold to and enhanced by Active Memory Technology, Inc. (AMT), Irvine, California. AMT's newest version, the DAP 610 system, has a performance 2.25 times that reported in the paper.

References

[1] M.L. Balinski, Signature methods for the assignment problem, *Oper. Res.* **33** (1985) 527–536.
[2] D.P. Bertsekas, A new algorithm for the assignment problem, *Math. Prog.* **6** (1981) 152–171.
[3] R.E. Burkard and U. Fincke, Probabilistic asymptotic properties of quadratic assignment problems, Report 81-3, Mathematisches Institut, Universität zu Koln, Fed. Rep. Germany, 1981.
[4] R.E. Burkard, Quadratic assignment problems, *European J. Oper. Res.* **15** (1984) 283–289.
[5] M.E. Dyer, A.M. Frieze and C.J.H. McDiarmid, Linear program with random costs, *Math. Prog.* **35** (1986) 3–16.
[6] M.J. Flynn, Very high-speed computing systems, *Proc. IEEE* **54** (1966) 1901–1909.
[7] L.R. Ford and D.R. Fulkerson, *Flows in Networks* (Princeton University Press, Princeton, NJ, 1962).
[8] C.C. Foster, *Content Addressable Parallel Processors* (Van Nostrand Reinhold, New York, 1976).
[9] A.M. Frieze and J. Yadegar, An algorithm for solving 3-dimensional assignment problems with application to scheduling a teaching practice, *J. Oper. Res. Soc.* **32** (1981) 989–995.
[10] M.R. Garey and D.S. Johnson, *Computers and Intractibility: A Guide to NP-Completeness* (Freeman, San Francisco, CA, 1979).
[11] B.W. Gostick, Software and algorithm for the Distributed Array Processor, *ICL Tech. J.* **1** (1979) 116–135.
[12] G.W.Graves and A.B. Whinston, An algorithm for the quadratic assignment problem, *Manag. Sci.* **17** (1970) 453–471.
[13] R.W. Hockney and C.R. Jesshope, *Parallel Computers* (Adam Hilger, Bristol, 1981).
[14] M.S. Hung and W.O. Rom, Solving the assignment problem by relaxation, *Oper. Res.* **28** (1980) 969–982.
[15] T.C. Koopmans and M. Beckmann, Assignment problems and the location of economic activities, *Econometrica* **25** (1957) 53–76.
[16] H.W. Kuhn, The Hungarian method for the assignment problems, *Naval Res. Logist. Quart.* **2** (1955) 83–97.
[17] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids* (Holt, Rinehart and Winston, New York, 1976).
[18] M. Los, Comparison of several heuristic algorithms to solve quadratic assignment problems of the Koopmans–Beckmans-type, Presented at the *International Symposium on Location Decisions*, Banff, Alberta, Canada, (1978).
[19] D. Parkinson, Practical parallel computers and their uses, in: D.J. Evans, ed., *Parallel Processing Systems* (Cambridge University Press, Cambridge, 1982) 215–235.
[20] D. Parkinson, The Distributed Array Processor (DAP), *Comput. Phys. Comm.* **28** (1983) 325–336.
[21] D. Parkinson, Organisational aspects of using parallel computers, *Parallel Comput.* **5** (1987) 75–83.
[22] S.F. Reddaway, The DAP approach, in: C.R. Jesshope and R.W. Hockney, eds., *Infotech State of the Art Report: Supercomputers, Vol. 2* (Infotech International, Maidenhead, 1979) 311–329.
[23] S. Sahni and T. Gonzalez, P-complete approximation problems, *J. ACM* **23** (1978) 555–565.
[24] L. Steinberg, The backboard wiring problem: A placement algorithm, *SIAM Rev.* **3** (1961) 37–50.