Parallel Computing 17 (1991) 707–732
North-Holland

Practical aspects and experiences

Parallel synchronous and asynchronous implementations of the auction algorithm *

Dimitri P. Bertsekas [a] and David A. Castañon [b]

[a] *Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, MA 02139, USA*
[b] *Department of Electrical, Computer and Systems Engineering, Boston University, Boston, MA 02215, USA*

Received November 1989
Revised July 1990, January 1991

Abstract

Bertsekas, D.P. and D.A. Castañon, Parallel synchronous and asynchronous implementations of the auction algorithm, Parallel Computing 17 (1991) 707–732.

In this paper we discuss the parallel implementation of the auction algorithm for the classical assignment problem. We show that the algorithm admits a totally asynchronous implementation and we consider several implementations on a shared memory machine, with varying degrees of synchronization. We also discuss and explore computationally the tradeoffs involved in using asynchronism to reduce the synchronization penalty.

Keywords. Assignment problem, auction algorithm; synchronous and asynchronous implementation; computational results; shared memory machines.

1. Introduction

We consider the classical problem of optimal assignment of n persons to n objects. Given a benefit a_{ij} that person i associates with object j, we want to find an assignment of persons to objects, on a one-to-one basis, that maximizes the total benefit. The auction algorithm, a method for solving this problem first proposed in [5], and subsequently developed and extended in [8–14] has been shown to be very effective in practice, particularly for sparse problems. The algorithm operates like an auction. There is a price for each object, and at each iteration, unassigned persons bid simultaneously for their 'best' objects thereby raising the corresponding prices. Objects are then awarded to the highest bidder. For a detailed presentation of the algorithm, we refer to [11].

* This work was supported in part by the Innovative Science and Technology Program of the Strategic Defense Initiative Office under the supervision of the Office of Naval Research, contract N00014-88-C-0718. The authors would like to thank the Mathematics and Computer Science Division of the Argonne National Laboratory for providing access to the Advanced Computer Research Facility and training in the use of the Encore Multimax.

The method is also well suited for implementation on parallel machines. There are two basic approaches here, as well as a third one that combines the first two. In the first approach, the bids of several unassigned persons are carried out in parallel, with a single processor assigned to each bid; we call this approach *Jacobi parallelization* in view of its similarity with parallel Jacobi methods for solving systems of equations. In the second approach, there is only one bid carried out at a time, but the calculation of the bids is done in parallel by several processors; we call this approach *Gauss–Seidel parallelization*. Finally, the third approach is a *hybrid* whereby multiple bids are carried out in parallel, and the calculation of each bid is shared by several processors. This third approach, with proper choice of the number of processors used for each parallel task, has the maximum speedup potential.

The auction algorithm is also a natural candidate for a totally asynchronous implementation, whereby the bid calculations may be done with out-of-date object price information and the highest bidder awards and subsequent price adjustments may be done with out-of-date bid information. The potential advantage of an asynchronous implementation is a reduction of the *synchronization penalty*. This is the delay incurred when several processors synchronize to calculate in parallel a single person bid, when several processors calculating separate person bids in parallel wait to make sure that up-to-date price information is available, and when the processors calculating in parallel the highest bidder awards wait for all bids to come in. Asynchronous algorithms are discussed in detail in [15], which gives many other references.

In this paper, we explore the merits of various synchronous and asynchronous implementations of the auction algorithm in a shared memory multiple instruction stream, multiple data stream (MIMD) parallel computer (the Encore Multimax). We prove the validity of an asynchronous implementation. Such a proof may also be inferred from the analysis of an asynchronous implementation of the ϵ-relaxation method [9,12], which contains the auction algorithm as a special case but can also solve general linear network problems. This inference is, however, very complex. The proof of this paper is based on first principles and is far simpler because it focuses on the assignment problem and is based on a less complex model of asynchronous computation.

In this paper we also compare a variety of synchronous and asynchronous implementations of the auction algorithm, in an effort to quantify the tradeoffs between Jacobi and Gauss–Seidel parallelization, as well as the effects of asynchronism. Our conclusion is that fairly substantial speedups (up to about 7 using a maximum of 16 processors) of the auction algorithm can be obtained on the Multimax, and that successful asynchronous implementations substantially outperform their synchronous counterparts. There have been several computational studies with parallel implementations of the auction algorithm as well as other assignment algorithms, but to our knowledge, the present paper is the first to report on the practical performance of asynchronous versions in a real parallel machine.

In particular, Kempa et al. [33] have reported on the parallel performance of various synchronous implementations of the auction algorithm on the Alliant FX/8 computer. They have experimented exclusively with dense problems and without using scaling. They implemented a synchronous hybrid algorithm which uses the vector processing capability of each of the Alliant's processors to scan the admissible objects for each bid, and uses multiple processors to process several bids in parallel. The Alliant FX/8 performs a lot of its synchronization in hardware, and therefore does not require the careful software synchronization which was used in our implementations on the Encore Multimax. For problems comparable to those of the size reported in this paper (e.g. 1000 person dense assignment problems, cost range [1, 1000]), Kempa et al. obtained total speedups of 8.578 for their hybrid auction algorithm using 8 vector processors. Such a speedup reflects the increased potential for Gauss–Seidel parallelism in dense problems and also the vector capability of each processor in the Alliant FX/8. Kempa et al. did not attempt to explain their overall speedup in terms of the speedup contributed by the vector processors and the speedup contributed by the multiple concurrent bids. Thus, it is not clear from their reported results whether an effective combination of Gauss–Seidel and Jacobi parallelization was occurring.

Castañon et al. [18] have studied the effectiveness of different synchronous implementations of the Gauss–Seidel auction algorithm, and the algorithm of Jonker and Volgenant [31] for solving dense and sparse assignment problems on different multiprocessor architectures. The latter algorithm is a two-phase method; the first phase is based on the relaxation method of [6] and [7], and is in fact the same as the auction algorithm with $\epsilon = 0$; the second phase is a sequential shortest path method. The work [18] illustrates the superiority of single instruction stream, multiple data stream (SIMD) architectures for achieving Gauss–Seidel parallelism, with demonstrated reductions in computation time (relative to the computation time on a single-processor Encore Multimax) in the order of 60 for assignment problems with 1000 persons. This work did not attempt to combine Gauss–Seidel and Jacobi parallelism for maximal speedup. Additional work on SIMD architecture was reported by Phillips and Zenios [39], and by Wein and Zenios [42] with synchronous implementations of a hybrid auction algorithm using ϵ-scaling on the Connection Machine CM-2 for dense problems.

Kennington and Wang [32] have reported on a parallel implementation of the Jonker and Volgenant algorithm [31] for dense assignment problems on the 8-processor Sequent Symmetry S81. In their implementation, multiple processors are used to construct shortest paths from a single unassigned person. This may be viewed as Gauss–Seidel parallelization for successive shortest path methods. For a dense 1000 person assignment problems with cost range [1, 1000], they report a speedup of 3.6 using 8 processors versus using a single processor.

Balas et al. [1] have developed a synchronous parallel successive shortest path algorithm, which allows for the determination of multiple augmenting paths simultaneously, and have successfully implemented it on a 14-processor Butterfly Plus computer. Their algorithm may be viewed as Jacobi parallelization for successive shortest path methods, since it handles multiple unassigned persons in parallel. For a comparable 1000 person dense assignment problem with cost range [1, 1000], they obtained a speedup of 2.21 for the successive shortest path part of their algorithm, and an overall speedup of 2.17 when compared to the sequential version of the algorithm implemented on the same computer. Larger speedups were obtained with much larger dense problems.

In the next Section we provide an overview of the auction algorithm and in Section 3 we define and prove the validity of the totally asynchronous version. In Section 4 we discuss general issues of parallel synchronous and asynchronous implementation, with an emphasis on shared memory machines and the Encore Multimax in particular. In Section 5 we discuss a variety of implementations and we report on the results of our computational tests.

2. The auction algorithm

In the assignment problem that we consider, n persons wish to allocate among themselves n objects, on a one-to-one basis. Each person i must select his/her object from a given subset $A(i)$. There is a given benefit a_{ij} that i associates with each $j \in A(i)$. An *assignment* is a set of k person–object pairs $(i_1, j_1), \ldots, (i_k, j_k)$, such that $0 \leqslant k \leqslant n$, $j_m \in A(i_m)$ for all m, and the persons $i_1, \ldots, i_k$ and objects $j_1, \ldots, j_k$ are all distinct. The total benefit of the assignment is the sum $\sum_{m=1}^{k} a_{i_m j_m}$ of the benefits of the assigned pairs. An assignment is called *complete* (or *incomplete*) if it contains $k = n$ (or $k < n$, respectively) person–object pairs. We want to find a complete assignment with maximum total benefit, assuming that there exists at least one complete assignment. This is the classical assignment problem, studied algorithmically by many authors [2–4,6,17,21,24,25,28–31,35,36,41], beginning with Kuhn's Hungarian method.

In the auction algorithm, each object j has a price p_j with the initial prices being arbitrary. Prices are adjusted upwards as persons 'bid' for their 'best' object, that is, the object for which the corresponding benefit minus the price is maximal. Only persons without an object submit a bid, and objects are awarded to their highest bidder.

In particular, the prices p_j are adjusted at the end of 'bidding' iterations. At the beginning of each iteration, we have a set of object prices and an incomplete assignment, and the algorithm terminates when a complete assignment is obtained. Each iteration involves a subset I of the persons that are unassigned at the beginning of the iteration. It has two phases:

Bidding phase.

Each person $i \in I$ determines an object $j_i \in A(i)$ for which $a_{ij} - p_j$ is maximized over j, i.e.

$$j_i = \arg \max_{j \in A(i)} \{a_{ij} - p_j\},$$

and submits a bid

$$p_{j_i} + \gamma_i$$

for this object, where γ_i is a positive bidding increment to be specified shortly.

Assignment phase.

Each object j that receives one or more bids, determines the highest of these bids, increases p_j to the highest bid, and gets assigned to the person who submitted the highest bid. The person that was assigned to j at the beginning of the iteration (if any) is now left without an object (and becomes eligible to bid at the next iteration). If an object does not receive any bid during an iteration, its price and assignment status are left unchanged.

It can be shown that if the bidding increments γ_i are bounded from below by some $\epsilon > 0$, this auction process terminates in a finite number of iterations with all persons having an object. To get a sense of this, note that if an object receives a bid in m iterations, its price must exceed its initial price by at least $m\epsilon$, while if an object is unassigned, its price has not yet changed from its initial value. Thus, for sufficiently large m, the object will become 'expensive' enough to be judged 'inferior' to some unassigned object by each person. It follows that there is a bounded number of iterations at which an object can be considered best and thus be preferred to all unassigned objects by some person. (This argument as stated, assumes that it is feasible to assign any person to any object but it can be generalized for the case where the set of feasible person–object pairs is limited, as long as there exists at least one feasible complete assignment; see e.g. [10,11,15].)

Whether the complete assignment obtained upon termination of the auction process is optimal depends strongly on the method for choosing the bidding increments γ_i. In a real auction, a prudent bidder would not place an excessively high bid for fear the object might be won at an unnecessarily high price. Consistent with this intuition, one can show that if the bidding increment γ_i is small enough to ensure that even after the bid is accepted, the object will be 'almost best' for the bidder, then the final assignment will be 'almost optimal'. In particular, we can show that if upon termination, we have

$$\max_j \{a_{ij} - p_j\} - \epsilon \leqslant a_{ij_i} - p_{j_i}, \quad \text{for all assigned pairs } (i, j_i), \tag{1}$$

(a property known as *ϵ-complementary slackness* or ϵ-CS for short), then the total benefit of the final assignment is within $n\epsilon$ of being optimal.

For a first principles derivation of this, note that the total benefit of *any* complete assignment $\{(i, j_i) \mid i = 1, \ldots, n\}$ satisfies

$$\sum_{i=1}^{n} a_{ij_i} \leqslant \sum_{j=1}^{n} p_j + \sum_{i=1}^{n} \max_j \{a_{ij} - p_j\},$$

for any set of prices $\{p_j \mid j=1,\dots,n\}$, since the second term of the right-hand side is no less than

$$\sum_{i=1}^{n} (a_{ij_i} - p_{j_i}),$$

while the first term is equal to $\sum_{i=1}^{n} p_{j_i}$. Therefore, the optimal total assignment benefit cannot exceed the quantity

$$A^* = \min_{\substack{p_j \\ j=1,\dots,n}} \left\{ \sum_{j=1}^{n} p_j + \sum_{i=1}^{n} \max_j \{a_{ij} - p_j\} \right\} \tag{2}$$

On the other hand, if the ϵ-CS property (1) holds upon termination of the auction process, then by adding Eq. (1) over all i, we see that

$$\sum_{i=1}^{n} \left(p_{j_i} + \max_j \{a_{ij} - p_j\} \right) \leqslant \sum_{i=1}^{n} a_{ij_i} + n\epsilon. \tag{3}$$

Since the left side above cannot be less than A^*, which as argued earlier, cannot be less than the optimal total assignment benefit, we see that the final total assignment benefit $\sum_{i=1}^{n} a_{ij_i}$ is within $n\epsilon$ of being optimal.

We note parenthetically, that the preceding derivation is guided by duality theory; the assignment problem can be formulated as a linear programming problem, and the minimization problem in the right side of Eq. (2) is a dual problem (see e.g. [11,15,20,38,40]).

Suppose now that the benefits a_{ij} are all integer, which is the typical practical case (if a_{ij} are rational, they can be scaled up to integer by multiplication with a suitable common positive integer). Then, the total benefit of any assignment is integer, so if $n\epsilon < 1$, a complete assignment that is within $n\epsilon$ of being optimal must be optimal. It follows, that *if*

$$\epsilon < \frac{1}{n},$$

the benefits a_{ij} are all integer, and the ϵ-CS condition (1) *is satisfied upon termination, then the assignment obtained is optimal.*

There is a standard method for choosing the bidding increments γ_i so as to maintain the ϵ-CS condition (1) throughout the auction process, assuming this condition is satisfied by the initial prices and the initial assignment (as is trivially the case when no objects are assigned initially). In this method, ϵ is a fixed positive number, and the bidding increment γ_i is given by

$$\gamma_i = \epsilon + v_i - w_i, \tag{4}$$

where v_i is the best object value,

$$v_i = \max_{j \in A(i)} \{a_{ij} - p_j\}, \tag{5}$$

attained for an object j_i, and w_i is the 'second best' object value

$$w_i = \max_{j \neq j_i, j \in A(i)} \{a_{ij} - p_j\}. \tag{6}$$

We will assume for convenience throughout that $A(i)$ contains at least two objects, so the maximum in Eq. (6) is well defined. This choice of the bidding increment is shown in *Fig. 1.* Note that we have $\gamma_i \geqslant \epsilon$, so based on the earlier argument, this choice guarantees termination of the auction algorithm. The ϵ-CS property is also maintained if γ_i has any value between ϵ and $\epsilon + v_i - w_i$. However, termination of the auction process is typically faster with the maximal choice of Eq. (4).

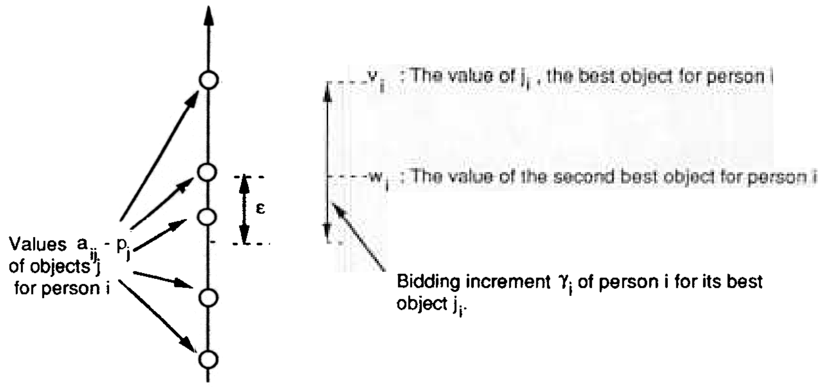
v_i : The value of j_i, the best object for person i
w_i : The value of the second best object for person i
ε
Values $a_{ij} - p_j$ of objects j for person i
Bidding increment γ_i of person i for its best object j_i.

Fig. 1. Illustration of the standard choice for the bidding increment γ_i. It is such that if the bid is accepted, the best object j_i will be within at most ϵ from being best. (It will be within exactly ϵ of being best if and only if at least one 'second best' object receives no bid during the iteration, so its price remains unchanged.)

Note that any nonempty subset I of unassigned persons may submit a bid at each iteration. This gives rise to a variety of possible implementations, named after their analogs in relaxation and coordinate descent methods for solving systems of equations or unconstrained optimization problems (see e.g. [37,15]):

(a) The *Jacobi* implementation, where I is the set of all unassigned persons at the beginning of the iteration.

(b) The *Gauss–Seidel* implementation, where I consists of a single person, who is unassigned at the beginning of the iteration.

(c) The *block Gauss–Seidel* implementation, where I is a subset of the set of all unassigned persons at the beginning of the iteration. The method for choosing the persons in the subset I may vary from one iteration to the next. This implementation contains the preceding two as special cases.

Generally, in a serial computation environment, experiments have shown that the Gauss–Seidel implementation tends to be the fastest, but with a parallel machine, the choice is unclear because all the bids of the persons in I may be calculated in parallel. It is important to consider all these different versions because they provide starting points for different synchronous and asynchronous parallel implementations, to be discussed in Section 4.

2.1. Computational aspects-ϵ-scaling

The auction algorithm exhibits interesting computational behavior and it is essential to understand this behavior in order to implement the algorithm efficiently.

We first note that the amount of work to solve the problem can depend strongly on the value of ϵ and on the maximum absolute object value

$$C = \max_{i,j} |a_{ij}|. \tag{7}$$

Basically, for many types of problems, the number of bidding iterations up to termination tends to be proportional to C/ϵ. We note also that there is a dependence on the initial prices; if these prices are 'near optimal', it can be expected that the number of iterations to solve the problem will be relatively small. This suggests the idea of *ϵ-scaling*, which consists of applying the algorithm several times, starting with a large value of ϵ and successively reducing ϵ up to an ultimate value which is less than the critical value $1/n$. Each application of the algorithm provides good initial prices for the next application. In practice, it is a good idea to consider scaling. For sparse assignment problems, that is, problems where the set of feasible assignment pairs is severely restricted, scaling seems almost universally helpful. This was established experimentally at the time of the original proposal of the auction algorithm [5]. There is also a related polynomial complexity analysis [12,15] that uses some of the earlier ideas of an ϵ-scaling analysis [26,27] for the ϵ-relaxation method of [9]. For more on this issue, we refer to [10] and [14], which contain extensive computational results.

Our implementation of ϵ-scaling is as follows: the integer benefits a_{ij} are first multiplied by $n+1$ and the auction algorithm is applied with progressively lower value of ϵ, up to the point where ϵ becomes 1 or smaller (because the a_{ij} have been scaled by $n+1$, it is sufficient for optimality of the final assignment to have $\epsilon \leqslant 1$). The sequence of ϵ values used is

$$\epsilon(k) = \lfloor \max\{1, \Delta/\theta^k\} \rfloor, \quad k = 0, 1, \ldots,$$

where Δ and θ are parameters set by the user with $\Delta > 0$ and $\theta > 1$. (In our implementations, we used $\Delta = C/4$ and $4 \leqslant \theta \leqslant 8$.)

3. The totally asynchronous version of the auction algorithm

One may view a synchronous parallel algorithm as a sequence of consecutive computation segments called *phases*. The computations within each phase are divided in some way among the processors of a parallel computing system. The computations of any two processors within each phase are independent, so the algorithm is mathematically equivalent to some serial algorithm. Phases are separated by *synchronization points*, which are times at which all processors have completed the computations of a given phase but no processor has yet started the computations of the next phase. In asynchronous parallel algorithms, the coordination of the computations of the processors is less strict. Processors are allowed to proceed with computations of a phase with data which may be out-of-date because the computations of the previous phase are incomplete. An asynchronous algorithm may contain some synchronization points but these are generally fewer than the ones of the corresponding synchronous version.

To get a first idea of the totally asynchronous implementation of the auction algorithm, it is useful to think of a person as an autonomous decision maker that obtains at unpredictable times information about the prices of the objects. Each unassigned person makes a bid a arbitrary times on the basis of its current object price information (that may be outdated because of communication delays). In a shared memory machine context, the role of the unassigned person is played by one or more processors that retrieve object prices from shared memory, and calculate a bid for the best object. There is asynchronism because the prices may have changed while the processors are calculating the bid.

We now formulate the totally asynchronous model, and we prove its validity. We denote

$p_j(t)$ = Price of object j at time t,

$r_j(t)$ = Person assigned to object j at time $t[r_j(t) = 0$ if object j is unassigned],

$U(t)$ = Set of unassigned persons at time $t[i \in U(t)$ if $r_j(t) \neq i$ for all objects $j]$.

We assume that $U(t)$, $p_j(t)$, and $r_j(t)$ can change only at *integer* times t; this involves no loss of generality, since t may be viewed as the index of a sequence of physical times at which events of interest occur.

In addition to $U(t)$, $p_j(t)$, and $r_j(t)$, the algorithm maintains at each time t, a subset $R(t) \subset U(t)$ of unassigned persons that may be viewed as having a 'ready bid' at time t. We assume that by time t, a person $i \in R(t)$ has used prices $p_j(\tau_{ij}(t))$ and $p_j(\bar{\tau}_{ij}(t))$ from some earlier (but otherwise arbitrary) times $\tau_{ij}(t)$ and $\bar{\tau}_{ij}(t)$ with $\tau_{ij}(t) \leqslant \bar{\tau}_{ij}(t) \leqslant t$ to compute the best value

$$v_i(t) = \max_{j \in A(i)} \{a_{ij} - p_j(\tau_{ij}(t))\}, \tag{8}$$

a best object $j_i(t)$ attaining the above maximum,

$$j_i(t) = \arg \max_{j \in A(i)} \{a_{ij} - p_j(\tau_{ij}(t))\}, \tag{9}$$

the second best value

$$w_i(t) = \max_{j \neq j_i(t), j \in A(i)} \{a_{ij} - p_j(\bar{\tau}_{ij}(t))\}, \tag{10}$$

and has determined a bid

$$\beta_i(t) = a_{ij_i}(t) - w_i(t) + \epsilon. \tag{11}$$

(Note that ordinarily the best and second best values should be computed simultaneously, which implies that $\tau_{ij}(t) = \bar{\tau}_{ij}(t)$. In some cases, however, it may be more natural or advantageous to compute the second best value after the best value, with more up-to-date price information, which corresponds to the case $\tau_{ij}(t) \leqslant \bar{\tau}_{ij}(t)$ for some pairs (i, j).)

The implication here is that unassigned persons i will enter the set $R(t)$ and become eligible to bid, following some computations which update $j_i(t)$ and $\beta_i(t)$. However, to maximize the generality and flexibility of our model, the precise mechanism by which these computations are done is left unspecified subject to the following two assumptions:

Assumption 1.

$$U(t)\text{: nonempty} \Rightarrow R(t')\text{: nonempty for some } t' \geqslant t.$$

Assumption 2. For all i, j, and t,

$$\lim_{t \to \infty} \tau_{ij}(t) = \infty.$$

Clearly an asynchronous auction algorithm cannot solve the problem if unassigned persons stop submitting bids and if old information is not eventually discarded. This is the motivation for the preceding two assumptions.

Initially, each person is assigned to at most one object, that is, $r_j(0) \neq r_{j'}(0)$ for all assigned objects j and j', and it will be seen that the algorithm preserves this property throughout its course. Furthermore, initially ϵ-CS holds, that is,

$$\max_{k \in A(i)} \{a_{ik} - p_k(0)\} - \epsilon \leqslant a_{ij} - p_j(0), \quad \text{if } i = r_j(0).$$

It will be shown shortly that this property is also preserved during the algorithm.

At each time t, if $R(t)$ is empty nothing happens. If $R(t)$ is nonempty the following occur:

(a) A nonempty subset $I(t) \subset R(t)$ of persons that have a bid ready is selected.

(b) Each object j for which the corresponding bidder set

$$B_j(t) = \{i \in I(t) \mid j = j_i(t)\} \tag{12}$$

is nonempty, determines the highest bid

$$b_j(t) = \max_{i \in B_j(t)} \beta_i(t) \tag{13}$$

and a person $i_j(t)$ for which the above maximum is attained

$$i_j(t) = \arg \max_{i \in B_j(t)} \beta_i(t). \tag{14}$$

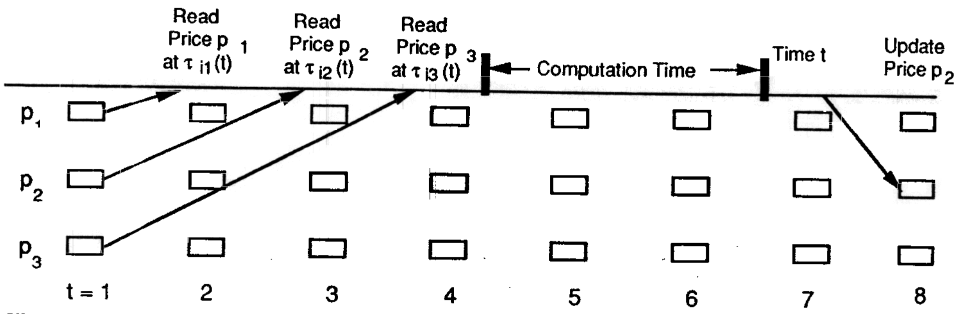
Read Price p 1 at τ i1 (t)
Read Price p 2 at τ i2 (t)
Read Price p 3 at τ i3 (t)
Computation Time
Time t
Update Price p2
p1
p2
p3
t = 1
2
3
4
5
6
7
8

Fig. 2. Illustration of asynchronous calculation of a bid by a single processor, which reads from memory the values p_j at different times $\tau_{ij}(t)$ and calculates at time t the best object

$$j_i(t) = \arg\min_{j \in A(i)} \{a_{ij} - p_j(\tau_{ij}(t))\},$$

and the maximum and second maximum values (here $\tau_{ij}(t) = \bar{\tau}_{ij}(t)$). The values of p_j may be out-of-date because they may have been updated by another processor between the read time $\tau_{ij}(t)$ and the bid calculation time t.

Then, the pair $(p_j(t), r_j(t))$ is changed according to

$$(p_j(t+1), r_j(t+1)) = \begin{cases} (b_j(t), i_j(t)) & \text{if } b_j(t) \geq p_j(t) + \epsilon \\ (p_j(t), r_j(t)) & \text{otherwise.} \end{cases}$$

The above description of the algorithm requires an infinite number of iterations; however, this is merely a mathematical convenience. In practice, the algorithm can be stopped as soon as the set of unassigned persons $U(t)$ is empty; this can be detected by counting the number of times that unassigned objects are assigned for the first time. We say that the algorithm *terminates at time t* if t is the first time k such that $U(k)$ is empty.

Notice that if $\tau_{ij}(t) = t$ and $U(t) = R(t)$ for all t, then the asynchronous algorithm is equivalent to the synchronous version given in Section 2. The asynchronous model becomes relevant in a parallel computation context where some processors compute bids for some unassigned persons, while other processors simultaneously update some of the object prices and corresponding assigned persons. Suppose that a single processor calculates a bid of person i by using the values $a_{ij} - p_j(\tau_{ij}(t))$ prevailing at times $\tau_{ij}(t)$ and then calculates the maximum value at time t; see *Fig. 2*. Then, if the price of an object $j \in A(i)$ is updated between times $\tau_{ij}(t)$ and t by some other processor, the maximum value will be based on out-of-date information. The asynchronous algorithm models this possibility by allowing $\tau_{ij}(t) < t$. A similar situation arises when the bid of person i is calculated cooperatively by several processors rather than by a single processor.

The following proposition establishes the validity of the asynchronous auction algorithm of this section.

Proposition 1. *Let Assumptions* 1 *and* 2 *hold and assume that there exists at least one complete assignment. Then for all t and all j for which* $r_j(t) \neq 0$, *the pair* $(p_j(t), r_j(t))$ *satisfies the* ϵ*-CS condition*

$$\max_{k \in A(i)} \{a_{ik} - p_k(t)\} - \epsilon \leq a_{ij} - p_j(t), \quad \text{if } i = r_j(t). \tag{16}$$

Furthermore, there is a finite time at which the algorithm terminates. The complete assignment obtained upon termination is within $n\epsilon$ *of being optimal, and is optimal if* $\epsilon < 1/n$ *and the benefits* a_{ij} *are integer.*

Proof. Let $(p_j(t), r_j(t))$ be a pair with $r_j(t) \neq 0$. To simplify notation, let $i = r_j(t)$. We first consider times t at which p_j was just updated, i.e., $p_j(t) > p_j(t-1)$ and $i \neq r_j(t-1)$, and person i submitted a highest bid for object j at time $t-1$. Then we have by construction

$$\begin{aligned} a_{ij} - p_j(t) &= a_{ij} - \beta_i(t-1) = w_i(t-1) - \epsilon \\ &= \max_{k \neq j, k \in A(i)} \{a_{ik} - p_k(\bar{\tau}_{ik}(t))\} - \epsilon \\ &\geqslant \max_{k \in A(i)} \{a_{ik} - p_k(t)\} - \epsilon, \end{aligned}$$

where the last inequality follows using the fact $p_k(t) \geqslant p_k(t')$ for all k and t, t' with $t \geqslant t'$. Therefore, the ϵ-CS condition (16) holds for all t at which p_j was just updated.

Next we consider times t for which p_j was not just updated. Let t' be the largest time which is less than t and for which $p_j(t') > p_j(t'-1)$; this is the largest time prior to t that object j was assigned to person i. By the preceding argument, we have

$$a_{ij} - p_j(t') \geqslant \max_{k \in A(i)} \{a_{ik} - p_k(t')\} - \epsilon,$$

and since $p_j(t') = p_j(t)$, and $p_k(t) \geqslant p_k(t')$ for all k, the ϵ-CS condition (16) again follows.

We next show that the algorithm terminates in finite time. We first note the following:

(a) Once an object is assigned, it remains assigned for the remainder of the algorithm (possibly to different persons). Furthermore, an unassigned object has a price equal to its initial price.

(b) Using Eqs. (8) and (10), and the relation $p_j(\tau_{ij}(t)) \leqslant p_j(\bar{\tau}_{ij}(t))$, which holds because $\tau_{ij}(t) \leqslant \bar{\tau}_{ij}(t)$, we have $a_{ij_i(t)} - p_{j_i(t)} \geqslant w_i(t)$, so from Eq. (11) we see that

$$\beta_i(t) \geqslant p_j(\tau_{ij_i}(t)) + \epsilon.$$

It follows from Eq. (13) that if person i bids for object j at time t, we must have

$$b_j(t) \geqslant p_j(\tau_{ij}(t)) + \epsilon. \qquad (17)$$

(c) Each time an object j receives a bid $b_j(t)$ at time t, there are two possibilities: either $b_j(t) < p_j(t) + \epsilon$, in which case $p_j(t+1) = p_j(t)$, or else $b_j(t) \geqslant p_j(t) + \epsilon$, in which case $p_j(t+1) \geqslant p_j(t) + \epsilon$ and $p_j(t)$ increases by at least ϵ [cf. Eq. (15)]. In the latter case we call the bid *substantive*. Suppose that an object receives an infinite number of bids during the algorithm. Then, an infinite subset of these bids must be substantive; otherwise $p_j(t)$ would stay constant for t sufficiently large, we would have $p_j(\tau_{ij}(t)) = p_j(t)$ for t sufficiently large because old price information is eventually purged from the system (cf. Assumption 2), and in view of Eqs. (15) and (17), we would have $p_j(t+1) \geqslant p_j(t) + \epsilon$ for all times t at which j receives a bid, arriving at a contradiction.

Assume now, in order to obtain a contradiction, that the algorithm does not terminate finitely. Then, because of Assumption 1, there is an infinite number of times t at which $R(t)$ is nonempty and at each of these times, at least one object receives a bid. Thus, there is a nonempty subset of objects J^∞ which receive an infinite number of bids, and a nonempty subset of persons I^∞ which submit an infinite number of bids. In view of (c) above, the prices of all objects in J^∞ increase to ∞, and in view of (a) above all objects in J^∞ are assigned to some person for t sufficiently large. Furthermore, the prices of all objects $j \notin J^\infty$ stay constant for t sufficiently large and since old information is purged from the system (cf. Assumption 2), we also have $p_j(\tau_{ij}(t)) = p_j(t)$ for all i, $j \notin J^\infty$, and t sufficiently large. These facts imply that for sufficiently large t, every object $j \in A(i)$ which is not in J^∞ would be preferable for person i to every object $j \in A(i) \cap J^\infty$. Since the ϵ-CS condition (1) holds throughout the algorithm, we see that for each person $i \in I^\infty$ we must have $A(i) \subset J^\infty$; otherwise such a person would bid for an object not in J^∞ for sufficiently large t.

We now note that after sufficiently long time, the only bids taking place will be by persons in I^∞ bidding for objects in J^∞, so each object in J^∞ will be assigned to some person from I^∞, while at least one person in I^∞ will be unassigned (otherwise the algorithm would terminate). We conclude that the number of persons in I^∞ is larger than the number of objects in J^∞. This, together with the earlier shown fact

$$A(i) \subset J^\infty, \quad \forall i \in I^\infty,$$

implies that there is no complete assignment, contradicting our problem feasibility assumption.

The optimality properties of the assignment obtained upon termination follow from the ϵ-CS property shown and our earlier discussion on the synchronous version of the algorithm. □

4. Synchronous and asynchronous implementations

In synchronous shared memory implementations of the auction algorithm, all bidding and assignment phases are separated by a synchronization point. There are two basic methods to parallelize the bidding phase for the set of unassigned persons I, and a third method which is a combination of the other two:

(a) *Parallelization across bids (or Jacobi parallelization).* Here the calculations involved in the bid of each person $i \in I$ are carried out by a single processor. If the number of persons in I, call it $|I|$, exceeds the number of processors p, some processors will execute the calculations involved in more than one bid. (This will typically happen in the early stages of a Jacobi-type algorithm where I is the set of all unassigned persons.) If $|I| < p$, then $p - |I|$ processors will be idle during the bidding phase, thereby reducing efficiency. (This will typically happen in the late stages of a Jacobi-type algorithm.)

(b) *Parallelization within a bid (or Gauss–Seidel parallelization).* Here the set I consists of a single person as in the Gauss–Seidel implementation. The calculations involved in the bid of each unassigned person i are shared by the p processors of the system. Thus the set of admissible objects $A(i)$ is divided in p groups of objects $A_1(i)$, $A_2(i), \ldots, A_p(i)$. The best object, best value, and second best value are calculated within each group in parallel by a separate processor. We call the calculations within a group a *search task*. After all the search tasks are completed (a synchronization of the processors is required to check this) the results are 'merged' by one of the processors who finds the best value over all best group values, while simultaneously computing the corresponding best object and size of bid. (It is possible to do the merging in parallel using several processors, but this is inefficient when the number of processors is small, as it was in our case, because of the extra synchronization and other overhead involved.) The drawback of this method over the preceding one is that it typically requires a larger number of iterations, since each iteration involves a single person. This is significant because even though each Gauss–Seidel iteration may take less time because it is executed by multiple processors in parallel, the synchronization overhead is roughly proportional to the number of iterations.

(c) *Hybrid approach (or block Gauss–Seidel parallelization).* In this approach, the bid calculations of each person are parallelized as in the preceding method, but the number of searcher processors used per bid is s, where $1 < s < p$. We will assume that s divides evenly p, so we can compute the bids of p/s persons in parallel, assuming enough unassigned persons are available for the iteration ($|I| \geqslant p/s$). With proper choice of s, this method combines the best features and alleviates the drawbacks of the preceding two.

Once the bidding phase of an iteration is completed (a synchronization point), the assignment phase is executed. This phase is carried out by a single processor in our synchronous implementations. While it is possible to consider using multiple processors to execute the assignment phase in parallel, the potential gain from parallelization is modest while the associated overhead more than offsets this gain in our system.

We have constructed an empirical model for the computation time per iteration of the block Gauss–Seidel method with p processors and s search tasks per bid. This time is given by

$$T(p, s) = S(p, s) + M(p, s) + C(p, s) + V,$$

where $S(p, s)$ is the time for completing the search tasks, $M(p, s)$ is the time for merging the results of search tasks, $C(p, s)$ is the time for synchronization and V is the constant overhead per iteration.

Let us assume for convenience that each set of admissible objects $A(i)$ has the same number of elements, say n. By counting the number of operations and by assuming perfect load balancing between the search tasks (i.e., an equal number of objects n/s in each of the groups $A_1(i), \dots, A_s(i)$), we have estimated roughly that the search time per iteration is

$$S(p, s) = \text{Const.} \cdot \left(\frac{n}{s} + \log \frac{n}{s} + \log\left(\log \frac{n}{s} \right) \right).$$

(The logarithmic terms account for the calculations involving the second best value.) The merging time is proportional to s,

$$M(p, s) = \text{Const.} \cdot s,$$

while the synchronization time was found experimentally to be roughly proportional to p

$$S(p, s) = \text{Const.} \cdot p;$$

see the next Section.

It can be seen that, given n, there are optimal values of p and s that minimize the total time per iteration. For example, if p and s are large, the increase of the synchronization and merging times may offset the potential gains from parallelization of the search tasks.

Another important consideration is that as p/s increases, the number of bids that can be calculated in parallel also increases, although not proportionally because near termination, the number of unassigned persons may be less than p/s. As a result, the number of iterations tends to decrease by a somewhat unpredictable factor, which is typically less than p/s. Because of this and because of various constants involved in the preceding estimates of the search, merging, and synchronization time, it is difficult to estimate a priori the optimal values of p and s to solve the problem. An interesting possibility that we did not try is to change dynamically s so that the number of unassigned persons is greater or equal to p/s throughout the algorithm.

4.1. An asynchronous implementation

In our asynchronous implementation, the bidding and merging calculations are divided in tasks, which are organized in a first in–first out queue. When a processor becomes free it starts executing the top task of the queue, if the queue is nonempty, and otherwise it checks whether a termination condition is satisfied. The algorithm stops when all processors encounter the termination condition.

Similarly as in the synchronous block Gauss–Seidel implementation, each set of admissible objects $A(i)$ is divided in s groups of objects $A_1(i), \dots, A_s(i)$. The calculation of the bid of a person i is divided in s tasks. The first $s-1$ tasks are search tasks involving the groups of objects $A_1(i), \dots, A_{s-1}(i)$. To perform one of these tasks, a processor must calculate and store in memory the best value, second best value, and best object within the corresponding object group. The sth task starts with a search and memory storage of the best value, second best value, and best object within the group $A_s(i)$, and following this, it completes the bid of person

i by merging the individual group search results, that is, by finding the best object and bid for person i based on the currently stored group results. The sth task also includes raising the price of the best object and changing the assignment of the object (assuming the calculated bid is larger than the best object's price by at least ϵ). An alternative is to create an extra task that changes the price and assignment of the objects; this leads, however, to an inefficient implementation as will be seen in the next Section.

There are two sources of asynchronism here. First, it is possible for some prices to be changed between the time a search task is completed and the time the results of that task are used to calculate a person bid. Second, it is possible that the merging task of a person's bid is carried out before some of the search tasks associated with that bid are completed. In both cases the bid may reflect out-of-date price information and may prove ineffective in that it yields a bid that does not exceed the corresponding best object's price by at least ϵ.

The advantage of the asynchronous implementation is that processors do not remain idle waiting to get synchronized with other processors or waiting for merging tasks to be completed.

The extreme special case of the preceding algorithm, where $s = 1$ and a person's bid is calculated by a single processor, is called *asynchronous Jacobi algorithm*. Generally one obtains more efficient implementations when $s > 1$, but the optimal value of s depends on the dimension and the sparsity structure of the problem.

5. Coded implementations and computational results

In this Section we describe the design and performance of six parallel auction algorithm implementations on the Encore Multimax. These implementations are:
(1) Synchronous Gauss–Seidel auction,
(2) Synchronous Jacobi auction,
(3) Synchronous hybrid auction,
(4) Asynchronous Jacobi auction,
(5) Asynchronous hybrid auction 1,
(6) Asynchronous hybrid auction 2.

We illustrate these algorithms by numerical experiments using a common 1000 person, 20% dense assignment problem with integer costs selected randomly in the range [1, 1000]. The size of the problem was large enough to allow for significant speedups using parallel processing. Additional numerical experiments with a variety of problem sizes have produced qualitatively similar results. A comparison of the synchronous and asynchronous auction versions is also given in this Section, based on solution of a broader range of problems.

5.1. Synchronous Gauss–Seidel auction algorithm

This algorithm processes a single bid at a time, by executing p search tasks in parallel, followed by merging the results of the search tasks, as discussed in the preceding Section. *Figure 3* shows that the one-processor version of the Gauss–Seidel auction algorithm spends a significant portion of its computation time (depending on the problem size and density) executing the search tasks. Thus, the algorithm has considerable speedup potential through parallelization of the search, particularly for dense problems.

The design of the synchronous Gauss–Seidel auction algorithm is illustrated in *Fig. 4*. Two synchronization points are included in each bidding iteration. The first is a barrier (based on the barrier monitor developed at ANL/MCS [16]), which serves to delay the start of the search of admissible objects until the previous price update is completed. The second synchronization point is an extension of the Argonne monitors for portable parallel programming [16]. It

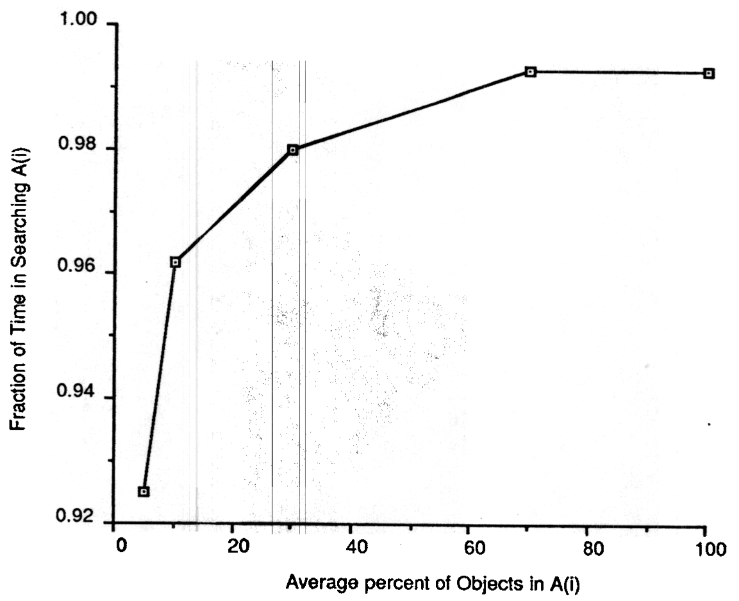

Fig. 3. Percentage of total computation time spent by the one-processor version of the Gauss–Seidel auction in searching the lists of admissible objects as a function of the density of feasible assignments, for 1000 person assignment problems, with cost range [1, 1000].

sequences the merging of the search task results and guarantees that the results of the merged search are identical with the one-processor Gauss–Seidel algorithm.

Figure 5 illustrates the performance of the synchronous Gauss–Seidel auction algorithm. All of the times reported in the figure are measured in terms of the parent processor (the processor which executes the sequential part of the algorithm). It is seen that the achievable speedup for the 1000 person, 20% dense problem is limited to about 3, because the synchronization and merging time increase with the number of processors at a rate slightly faster than linear. Generally, for a fixed number of processors, the speedup of the synchronous Gauss–Seidel auction typically increases as the problem density increases, since then the serial time for searching (which is parallelized) increases relative to the serial time for merging (which is not parallelized), as well as the time for synchronization.

Figure 6 illustrates the conjectured theoretical behavior of the total search, synchronization and computation times, based on fitting the models described in the previous Section with

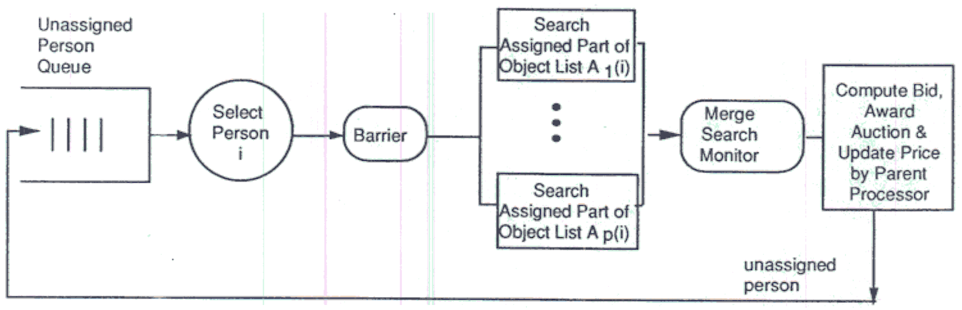
Unassigned Person Queue
Select Person i
Barrier
Search Assigned Part of Object List A 1(i)
Search Assigned Part of Object List A p(i)
Merge Search Monitor
Compute Bid, Award Auction & Update Price by Parent Processor
unassigned person

Fig. 4. Design of the parallel synchronous Gauss–Seidel auction algorithm. Multiple processors are used to search the list of admissible objects for a person; the results of the searches are merged to compute a person's bid, and the price and object assignment update is done by a single processor.

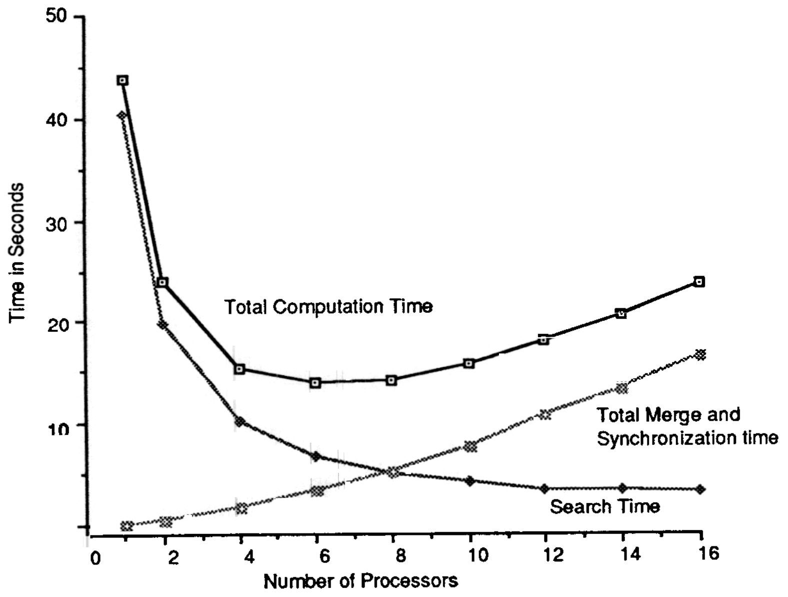

Fig. 5. Performance of the synchronous Gauss–Seidel auction algorithm as the number of processors increases for a 1000 person, 20% dense assignment problem with cost range [1, 1000]. Note the growth in the merging and synchronization time as the number of processors increases. This limits the overall speedup to approximately 3 for this problem.

appropriate constants to match the problem size. Note the close correspondence between the predictions of *Fig. 6* and the empirical results of *Fig. 5*. The only discrepancy is that the empirical synchronization time grows slighly faster than the predicted time with the number of

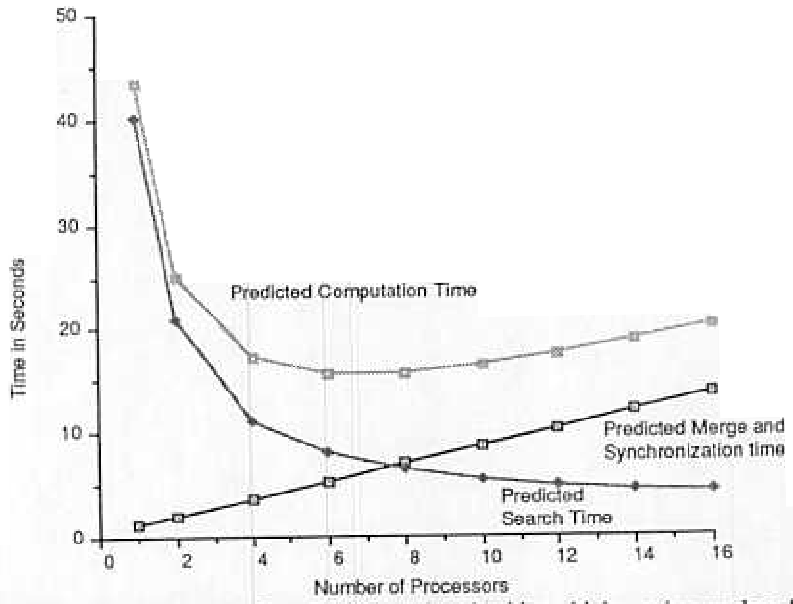

Fig. 6. Theoretical behavior of synchronous Gauss–Seidel auction algorithm with increasing number of processors. The constants have been matched to fit the times of a 1000 person, 20% dense assignment problem with cost range [1, 1000].

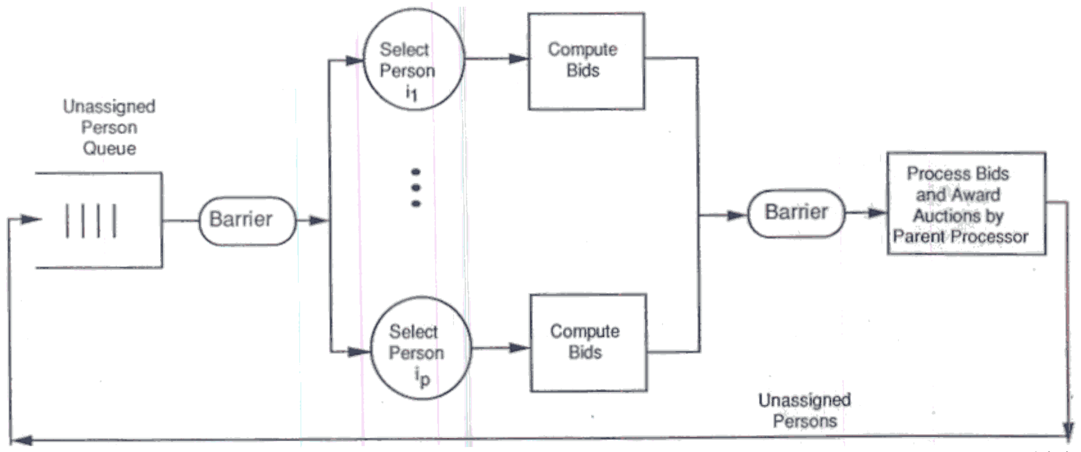

Fig. 7. Design of synchronous Jacobi auction algorithm. Multiple processors are used to compute bids for multiple persons simultaneously. The parent processor then processes sequentially the bids.

processors; this is probably due to increased contention for access to critical sections in the monitors. Similar phenomena were observed by Dritz and Boyle [22] in their experiments using the Encore Multimax.

5.2. Synchronous Jacobi auction algorithm

In this algorithm, multiple processors are used to generate bids simultaneously for different persons. The number of simultaneous bids is equal to the minimum of the number of processors used and the number of unassigned persons. Each processor computes the bid associated with a different person. The resulting bids are then processed at a single processor, called the parent, in order to update the object prices and assignments, and the list of unassigned persons. The design of the algorithm is illustrated in *Fig. 7*. Again, there are two synchronization points per iteration, which are implemented with the extension of the barrier monitor discussed previously. The synchronization after the compute bids operation is only a barrier monitor because no merging of the individual computations by each processor is required (unlike the synchronous Gauss–Seidel auction algorithm). It turns out that this reduces the overall synchronization overhead.

An important aspect of the synchronous Jacobi auction algorithm is that the amount of potential parallel work varies across iterations; specifically, it depends on the number of remaining unassigned persons. When this number is less than the number of available processors, some of the processors will be idle; see *Fig. 8*. In order to prevent idle processors for competing for shared resources such as synchronization locks, the size of the synchronization barriers was adaptively modified to match the number of non-idle processors. Idle processors were diverted to a rest barrier, waiting to rejoin the computation when the number of unassigned persons grew larger than the number of available processors (at the beginning of a new ϵ-scaling phase).

Figure 9 illustrates the performance of the synchronous Jacobi auction algorithm. Again, search time and synchronization time were measured for the parent processor. The search time per iteration is independent of the number of processors, but the total number of iteration (and therefore also the total search time) is reduced when the number of processors increases because then the average number of parallel bids per iteration also increases. Note the relatively small synchronization time required for the Jacobi auction algorithm when compared to the

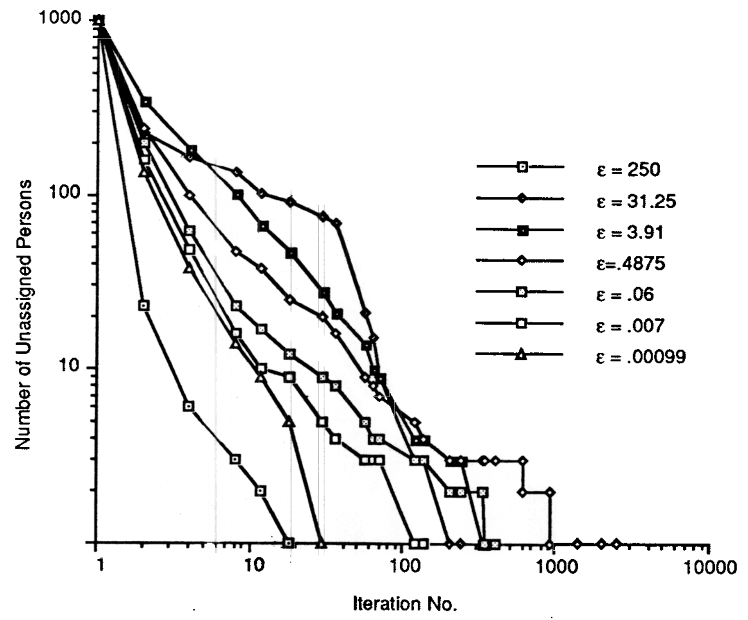

Fig. 8. Number of unassigned persons versus iteration number in Jacobi auction using 10 processors, for a 1000 person, 20% dense problem, with cost range [1, 1000]. Curves illustrate the number of unassigned persons for different values of ϵ corresponding to different ϵ-scaling phases. Note that for many iterations, the number of unassigned persons exceeds the number of available processors, resulting in loss of efficiency.

Gauss–Seidel algorithm. This is due to three factors. First, the synchronization after computing bids is simpler because no merging of the results of the processors is required. Second, the number of synchronization calls is reduced because the total number of iterations is reduced by

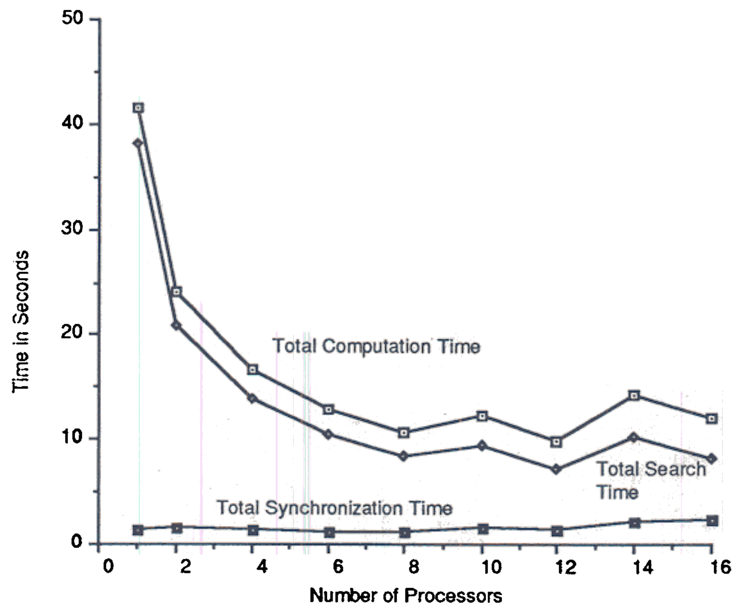

Fig. 9. Performance of the synchronous Jacobi auction algorithm for a 1000 person, 20% dense assignment problem, cost range [1, 1000] as a function of the number of processors.

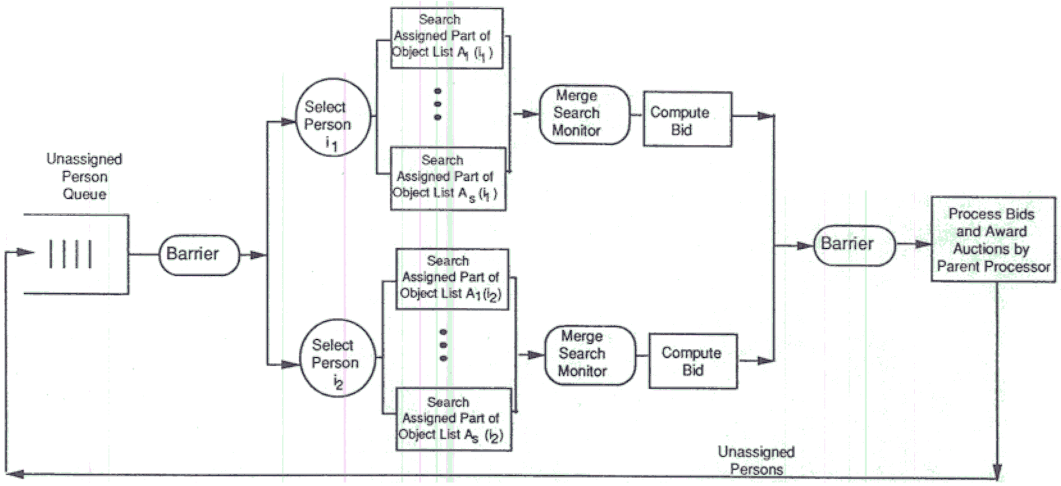
Unassigned Person Queue
Barrier
Select Person i_1
Select Person i_2
Search Assigned Part of Object List $A_1(i_1)$
Search Assigned Part of Object List $A_s(i_1)$
Search Assigned Part of Object List $A_1(i_2)$
Search Assigned Part of Object List $A_s(i_2)$
Merge Search Monitor
Compute Bid
Merge Search Monitor
Compute Bid
Barrier
Process Bids and Award Auctions by Parent Processor
Unassigned Persons

Fig. 10. Design of the synchronous hybrid auction algorithm with two bids per iteration, and $s = p/2$ search tasks per bid.

processing multiple bids in parallel. Finally, the number of processors which contend for a synchronization lock is reduced adaptively when the number of unassigned persons is less than the number of processors, leading to simpler synchronization (with reduced contention) at each iteration.

The results of *Fig. 9* reflect a small anomaly: increasing the number of processors from 8 to 10 produces an apparent increase in computation time. The reason is that, due to accidental reasons, the number of iteration required for convergence with 10 processors increased significantly over the corresponding number with 8 processors (the sample path of the algorithm changes with the number of processors).

5.3. Synchronous hybrid auction algorithm

The results obtained with the previous two synchronous algorithms suggest that an efficient parallel implementation should combine the speedups available from Gauss–Seidel parallelization and Jacobi parallelization. In particular, by computing multiple bids simultaneously, and by using multiple processors to compute each bid, a multiplicative effect may be achievable whereby the overall speedup is the product of the Gauss–Seidel speedup and the Jacobi speedup. The synchronous hybrid auction algorithm is an attempt to realize this multiplicative speedup. In this algorithm, unassigned persons are selected two at a time, and two bids are computed in parallel (Jacobi parallelization with two processors). For each person i, the set of admissible objects $A(i)$ is searched in parallel by $p/2$ processors (Gauss–Seidel parallelization).

The overall design of the algorithm is illustrated in *Fig. 10*. There are three synchronization points per iteration. An initial barrier is included to delay the start of the search tasks until all of the object prices are updated from the previous iteration. A separate merge search monitor is included for each person, and a synchronization barrier is used to wait until both bids are computed before proceeding to award the auctions. The size of the barriers and monitors were tailored to the number of processors which rendezvous at each synchronization point. Thus, the first barrier synchronizes $2s$ processors, the merge search monitors synchronize s processors, and the last barrier synchronizes only two processors, thereby keeping the synchronization overhead to a minimum.

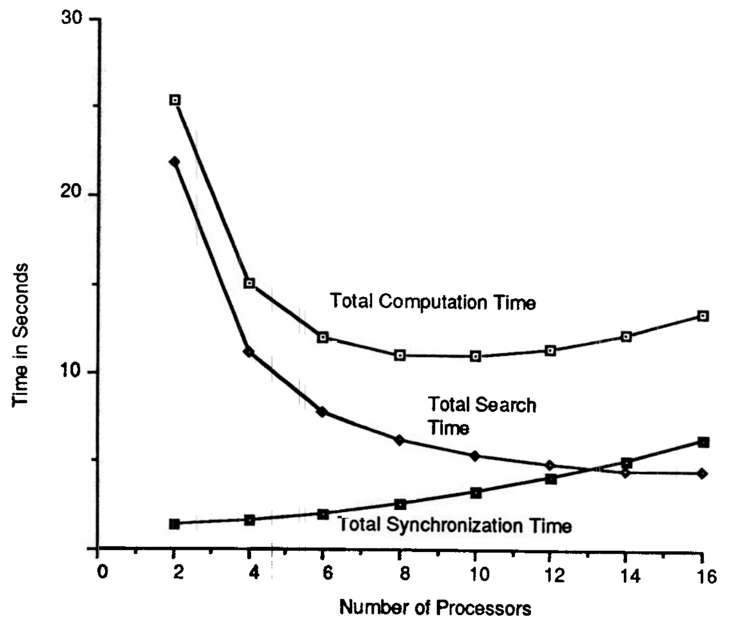

Fig. 11. Performance of the synchronous hybrid auction algorithm as a function of the number of processors for 1000 person, 20% dense assignment problem, cost range [1, 1000].

Figure 11 illustrates the performance of the synchronous hybrid auction algorithm as a function of the total number of processors used for the same 1000 person, 20% dense assignment problem described previously. The one-processor time for this algorithm is 44 seconds. The synchronization time is again measured in terms of the parent processor, and represents the total time that the parent processor spends at the different synchronization points. The curves in *Fig. 11* indicate that the achieved speedup is considerably lower than the anticipated multiplicative speedup from combining the Jacobi and Gauss–Seidel speedups. For example from *Fig. 11*, the actual speedup using 12 processors is under 4. If we multiply the speedup from Jacobi parallelization with two bids (which is roughly 1.75 based on *Fig. 9*), and the speedup from Gauss–Seidel parallelization using 6 processors (which is 2.75 based on *Fig. 5*), we obtain a predicted speedup of 4.8125. This loss of effectiveness can be traced to the growth of the synchronization time with the total number of processors used (even though the total number of iterations has been reduced by a factor of 1.83 due to Jacobi parallelization). This synchronization time represents the dominant part of the overall computation time when the number of processors is large, and prevents a multiplicative combination of the speedups from Gauss–Seidel and Jacobi parallelization.

5.4. Asynchronous Jacobi auction algorithm

This algorithm tries to reduce the overall synchronization overhead by allowing bids to be computed based on older values of the object prices. Specifically, processors start computing new bids without waiting for other processors to complete their price updates. Some synchronization is still required to guarantee that the object prices are monotonically increasing (cf. Eq. (15)), and to guarantee that the computation of a person bid is not unnecessarily replicated by multiple processors. This synchronization is implemented using locks on each object and a lock on the queue of unassigned persons; these locks allow only one processor at a time to modify the price of a given object, and only one processor at a time to update the queue of unassigned

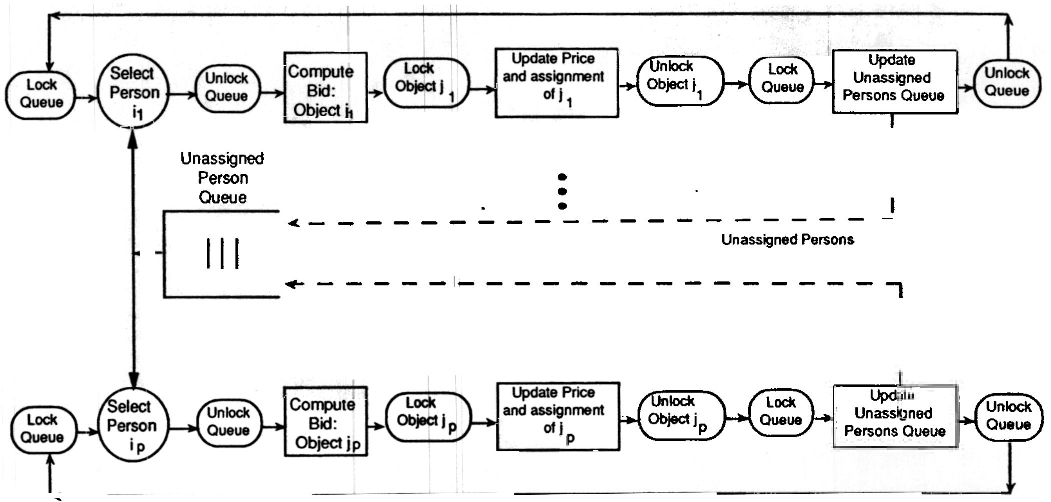

Fig. 12. Design of asynchronous Jacobi auction algorithm. Locks on each object and on the unassigned persons queue are used to guarantee data integrity and preserve complementary slackness.

persons. *Figure 12* illustrates the design of the asynchronous Jacobi auction algorithm. To reduce contention for the locks when the number of persons in the unassigned persons queue is lower than the number of processors, excess processors are diverted to a barrier to wait for a new ϵ-scaling cycle.

The performance of the asynchronous Jacobi auction algorithm is illustrated in *Fig. 13* for three sizes of randomly generated problems: 500 persons with 80% density, 1000 persons with 20% density and 2000 persons with 5% density. The problem densities were selected to obtain

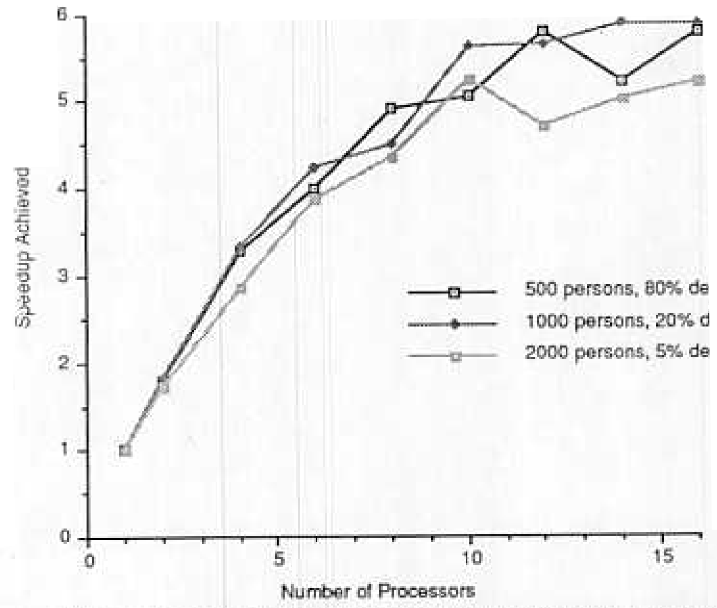

Fig. 13. Performance of the asynchronous Jacobi auction algorithm for assignment problems of different sizes: 500 person, 80% dense, 1000 person, 20% dense and 2000 person, 5% dense assignment problems, cost range [1, 1000]. The speedups shows are the average over three runs.

nearly equal sequential run times for all three problems. The curves in *Fig. 13* show the speedup over the sequential processing time obtained by the asynchronous Jacobi auction algorithm as a function of the number of processors used. The numbers shown represent an average of three runs; the actual running time of the algorithm varies from run to run because the order in which different processors complete their bids and acquire the locks affects the order in which objects are inserted into the unassigned persons queue. A different ordering of persons produces a different auction process, which affects the number of bids which must be generated for convergence. In the test runs, the number of bids generated varied by under 4% from run to run.

Note the increase in speedup achieved by the asynchronous Jacobi auction algorithm compared to the results obtained for similar problems by the synchronous Jacobi auction algorithm in *Fig. 9*; the speedups have been improved from 4.5 to nearly 5.8, which represents a 29% improvement. The increased speedup is achieved because of the improved load balance among the processors, as processors do not wait idly for other processors to complete their bidding process.

Another important point illustrated by *Fig. 13* is the effect of problem size on the speedup achievable through Jacobi parallelization. Note that the speedups obtained for all three problem sizes are roughly comparable; the fluctuations in speedup when using large numbers of processors are due to variations in the number of bids required for convergence when different numbers of processors are used. The reason for this behavior is that, although more bids are generated for larger problems, the number of iterations for which there are few bidders (e.g. 1 or 2) also increases for large problems, thereby limiting the potential speedup. In contrast, the potential speedup achievable through Gauss–Seidel parallelization increases with problem size, as the number of feasible assignments for each person increases.

ıment

ence
rent
17
lem
ms.

5.5. *Asynchronous hybrid auction algorithms*

We implemented two asynchronous hybrid auction algorithms. One of the two algorithms is quite inefficient, but the reasons for this are worth explaining. The design of the algorithms is illustrated in *Fig. 14*. Instead of an unassigned person queue, there is a queue of unassigned

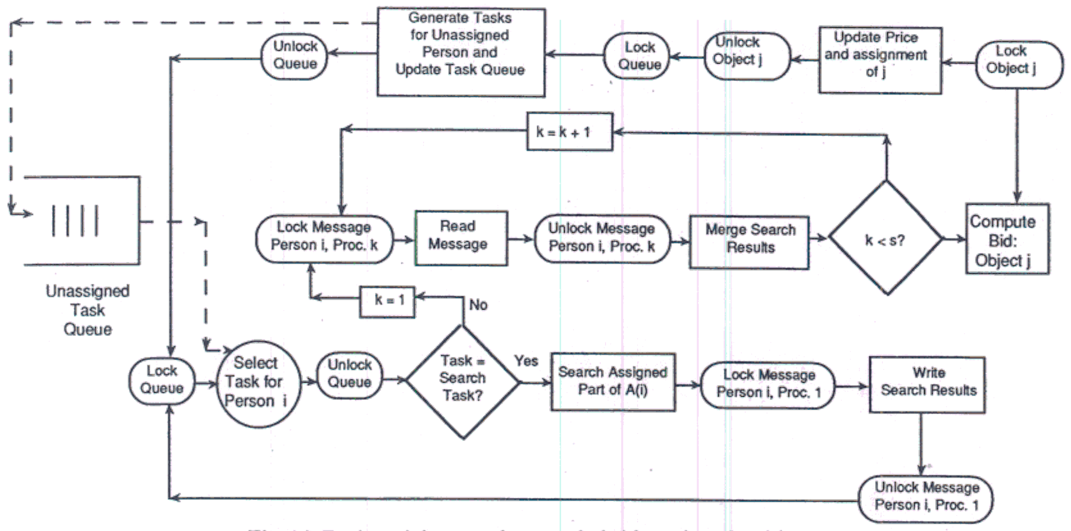
Generate Tasks for Unassigned Person and Update Task Queue
Unlock Queue
Lock Queue
Unlock Object j
Update Price and assignment of j
Lock Object j
k = k + 1
Lock Message Person i, Proc. k
Read Message
Unlock Message Person i, Proc. k
Merge Search Results
k < s?
Compute Bid: Object j
Unassigned Task Queue
k = 1
No
Lock Queue
Select Task for Person i
Unlock Queue
Task = Search Task?
Yes
Search Assigned Part of A(i)
Lock Message Person i, Proc. 1
Write Search Results
Unlock Message Person i, Proc. 1

Fig. 14. Design of the asynchronous hybrid auction algorithms.

n
r

search tasks and bid tasks. Each unassigned person is represented by s search tasks and one bid task, ordered consecutively in the queue so that the bid task follows the s search tasks. Different types of asynchronous algorithms can be generated by controlling the number of search tasks generated for each unassigned person. As before, a synchronization lock is required to allow tasks to be read and generated one at a time.

Figure 14 illustrates the processing of a single processor. After reading a task from the task queue, the processor determines whether it is a search task or a bid task. If it is a search task for person i, the processor searches the appropriate segment of the object set $A(i)$ and writes a message in shared memory with the results of its search (the two highest net profit levels and the object offering the highest net profit). The message is protected by a lock indexed by the processor index and the person index, which guarantees that the message must be read in its entirety by the bid task. After writing the message, the processor releases the lock and attempts to acquire another task.

If the task acquired is a bid task, the processor must read the message left by the search tasks for this person. Some of these search tasks may still by in process, so the bid processor may be reading old messages. The processor locks each message, reads the contents, releases the lock, and merges the results of the individual search tasks into an overall search result. This is then used to compute a bid (from person i to object j). The processor then locks object j, updates the price and assignment of object j, and releases the object. If an unassigned person results from this operation, the processor then locks the unassigned task queue, inserts s search tasks and one bid task for the unassigned person at the end of the queue, and releases the queue.

The algorithm described above will be called asynchronous hybrid auction 1 (or AHA1) algorithm. The difficulty with this algorithm is that a bid is often computed based on outdated information, leading to a large increase in the number of losing bids (and therefore the number of iterations required for convergence). Ideally, the bid task for person i would wait for the search tasks for person i to be completed; however, this requires time-wasting synchronization. An alternative way to accomplish the same effect is to require the processor that executes the last search task corresponding to a person to also execute the bid task corresponding to that person immediately after the search task. In this manner, the likelihood that the other search tasks corresponding to that person are complete by the time the bid task is executed is substantially increased. We call this version of the algorithm asynchronous hybrid auction 2 (or AHA2).

Figure 15 illustrates the relative performance (averaged across three runs) of the AHA1 and AHA2 algorithms for the same 1000 person assignment problem. Here the number of search tasks per bid was two for all the runs reported. Thus, the overhead for merging the search results is independent of the number of processors used. The AHA1 algorithm is nearly twice as slow as the AHA2 algorithm. The reason is illustrated in *Fig. 16*, which shows the number of bids generated by each algorithm up to convergence. The number of bids for the AHA1 algorithm is nearly double! The explanation is that the bid task is often generating the bids before the search tasks have been completed; these bids based on old information are often rejected, so that additional bids are required. The results illustrate the importance of careful management of asynchronous tasks in order to guarantee that the processors are doing useful work (i.e. work that will not become irrelevant when new information is acquired.)

Figure 17 illustrates the performance of one variation of the AHA2 algorithm for several 1000 person, 20% dense assignment problems with different structure. Three classes of problems were used in these experiments: randomly generated problems with cost range [1, 1000], symmetric cost problems with cost range [1, 1000] and extended cost problems with 'difficult' cost structure, where each cost element is selected randomly from the range [1, $i \times j$] (where i, j are the person and object indices, respectively). In these experiments, the number of

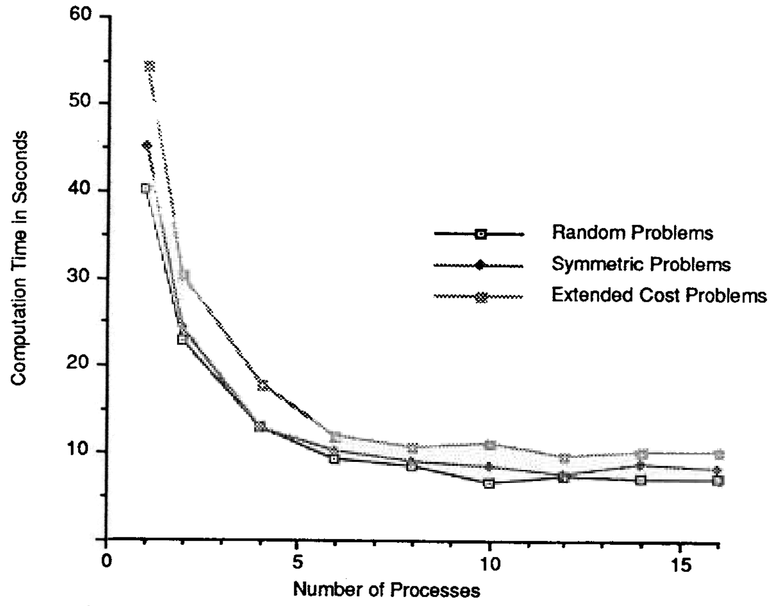

Fig. 17. Average computation time of asynchronous hybrid auction 2 for different 1000 person, 20% dense assignment problems: random problems, symmetric problems and extended cost problems with 'difficult' structure. The times shown are the average of three different runs. In these problems, the number of search tasks per bid was equal to 2.

Figure 17 illustrates that the AHA2 algorithm obtains similar reductions in computation time for each problem class using parallel processing.

We finally compare in *Fig. 18*, the performance of the synchronous and the asynchronous

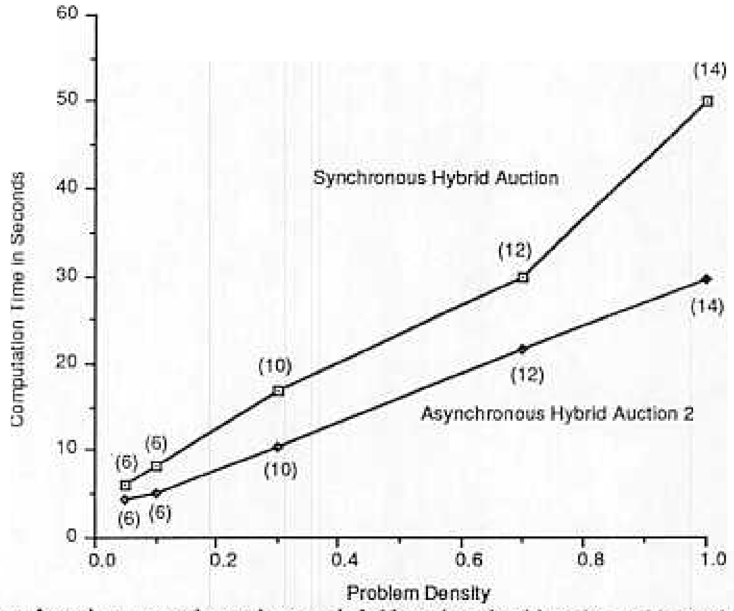

Fig. 18. Comparison of synchronous and asynchronous hybrid auction algorithms for problems with 1000 persons, cost range [1, 1000], and varying density. An 'optimal' number of processors was used for each run. This number is given in parenthesis next to the corresponding data point. The number of search tasks per bid in both algorithms was equal to half the number of processors.

hybrid auction algorithms on several problems, keeping the number of persons constant and varying the problem density. The asynchronous algorithm is the more efficient AHA2 version; the number of search tasks per bid is equal to half the number of processors in each algorithm. The benefits of asynchronism are clear from the figure.

6. Conclusions

In this paper, we have proved the validity of an asynchronous version of the auction algorithm, which can serve as a model of asynchronous implementations in a broad variety of parallel machines. We have also provided the first experimental comparison of a variety of synchronous and asynchronous versions of the algorithm. Our conclusion is that the better asynchronous implementations outperform substantially the corresponding synchronous implementations on a shared memory machine. This is in agreement with other recent studies [19,23], which have confirmed the advantage of asynchronous implementations of parallel network flow algorithms.

References

[1] E. Balas, D. Miller, J. Pekny and P. Toth, A parallel shortest path algorithm for the assignment problem, Management Science Report MSRR 552, Carnegie Mellon Univ., Pittsburgh, PA, April 1989.

[2] M.L. Balinski, Signature methods for the assignment problem, *Oper. Res.* 33 (1985) 527–537.

[3] M.L. Balinski, A competitive (dual) simplex method for the assignment problem, *Math. Program.* 34 (1986) 125–141.

[4] R. Barr, F. Glover and D. Klingman, The alternating basis algorithm for assignment problems, *Math. Program.* 13 (1977) 1–13.

[5] D.P. Bertsekas, A distributed algorithm for the assignment problem, Lab. for Information and Decision System Working Paper, M.I.T., March 1979.

[6] D.P. Bertsekas, A new algorithm for the assignment problem, Math. Program. 21 (1981) 152–171.

[7] D.P. Bertsekas, A unified framework for minimum cost network flow problems, Laboratory for Information and Decision Report LIDS-P-1245-A, M.I.T., Cambridge, MA, 1982; also in *Math. Program.* 32 (1985) 125–145.

[8] D.P. Bertsekas, A distributed asynchronous relaxation algorithm for the assignment problem, *Proc. 24th IEEE Conf. on Dec. and Control*, 1985, 1703–1704.

[9] D.P. Bertsekas, Distributed asynchronous relaxation methods for linear network flow problems, LIDS Report P-1606, M.I.T., Nov. 1986.

[10] D.P. Bertsekas, The auction algorithm: a distributed relaxation method for the assignment problem, *Ann. Oper. Res.* 14 (1988) 105–123.

[11] D.P. Bertsekas, *Linear Network Optimization: Algorithms and Codes* (MIT Press, Cambridge, MA, 1991).

[12] D.P. Bertsekas and J. Eckstein, Dual coordinate step methods for linear network flow problems, Laboratory for Information and Decision Systems Report LIDS-P-1768, M.I.T., Cambridge, MA, 1988, also in *Math. Progr.* 42 (1988) 203–243.

[13] D.P. Bertsekas and D.A. Castañon, The auction algorithm for transportation problems, *Ann. Oper. Res.* 20 (1989) 67–96.

[14] D.P. Bertsekas and D.A. Castañon, The auction algorithm for the minimum cost network flow problem, Laboratory for Information and Decision Systems Report LIDS-P-1925, M.I.T., Cambridge, MA, Nov. 1989.

[15] D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods* (Prentice-Hall, Englewood Cliffs, NJ, 1989).

[16] J. Boyle, R. Butler, T. Disz, B. Glickfield, E. Lusk, R. Overbeek, J. Patterson and R. Stevens, *Portable Programs for Parallel Processors* (Holt, Rinehart and Winston, New York, 1987).

[17] G. Carpaneto, S. Martello and P. Toth, Algorithms and codes for the assignment problem, *An. Oper. Res.* 13 (1988) 193–223.

[18] D. Castañon, B. Smith and A. Wilson, Performance of parallel assignment algorithms on different multiprocessor architectures, Argonne National Lab. Report, in preparation.

[19] E.D. Chajakis and S.A. Zenios, Synchronous and asynchronous implementations of relaxation algorithms for nonlinear network optimization, Report 89-10-07, Dept. of Decision Sciences, The Wharton School, Univ. of Pennsylvania, Phil., PA, October 1989.

[20] G.B. Dantzig, *Linear Programming and Extensions* (Princeton Univ. Press, Princeton, NJ, 1963).
[21] U. Derigs, The shortest augmenting path method for solving assignment problems – motivation and computational experience, *Ann. Oper. Res.* 4 (1985) 57–102.
[22] K.W. Dritz and J.M. Boyle, Beyond "speedup": performance analysis of parallel programs, Argonne National Lab. Report ANL-87-7, Feb. 1987.
[23] Didier El Baz, A computational experience with distributed asynchronous iterative methods for convex network flow problems, *Proc. 28th Conf. on Decision and Control*, Tampa, FL, Dec. 1989.
[24] M. Engquist, A successive shortest path algorithm for the assignment problem, *INFOR* 20 (1982) 370–384.
[25] F. Glover, R. Glover and D. Klingman, Threshold assignment algorithm, Center for Business Decision Analysis Report CBDA 107, Graduate School of Business, Univ. of Texas at Austin, Sept. 1982.
[26] A.V. Goldberg, Efficient graph algorithms for sequential and parallel computers, Tech. Report TR-374, Laboratory for Computer Science, M.I.T., Feb. 1987.
[27] A.V. Goldberg and R.E. Tarjan, Solving minimum cost flow problems by successive approximation, *Proc. 19th ACM STOC*, May 1987.
[28] D. Goldfarb, Efficient dual simplex methods for the assignment problem, *Math. Program.* 33 (1985) 187–203.
[29] M. Hall Jr., An algorithm for distinct representative, *Amer. Math. Monthly* 51 (1956) 715–717.
[30] M. Hung, A polynomial simplex method for the assignment problem, *Oper. Res.* 31 (1983) 595–600.
[31] R. Jonker and A. Volgenant, A shortest augmenting path algorithm for dense and sparse linear assignment problems, *Computing* 38 (1987) 325–340.
[32] J. Kennington and Z. Wang, Solving dense assignment problems on a shared memory multiprocessor, Techn. Report 88-OR-16, Dept. of Operations Research and Applied Science, Southern Methodist University, Oct. 1988.
[33] D. Kempa, J. Kennington and H. Zaki, Performance characteristics of the Jacobi and Gauss–Seidel versions of the auction algorithm on the Alliant FX/8, Report OR-89-008, Dept. of Mech. and Ind. Eng., Univ. of Illinois, Champaign-Urbana, 1989.
[34] H.W. Kuhn, The Hungarian method for the assignment problem, *Nav. Res. Logist. Q.* 2 (1955) 83–97.
[35] L.F. McGinnis, Implementation and testing of a primal–dual algorithm for the assignment problem, *Oper. Res.* 31 (1983) 277–291.
[36] J. Munkres, Algorithms for the assignment and transportation problems, *SIAM J.* (1956).
[37] J.M. Ortega and W.C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables* (Academic Press, NY, 1970).
[38] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity* (Prentice-Hall, Englewood Cliffs, NJ, 1982).
[39] C. Phillips and S.A. Zenios, Experiences with large scale network optimization on the Connection Machine, Report 88-11-05, Dept. of Decision Sciences, The Wharton School, Univ. of Pennsylvania, Phil., Penn., Nov. 1988; also in *Impact of Recent Computer Advances on Operations Research, Publ. Oper. Res. Ser.* 9, 164–180.
[40] R.T. Rockafellar, *Network Flows and Monotropic Programming* (Wiley-Interscience, NY, 1984).
[41] G.L. Thompson, A recursive method for solving assignment problems, in: P. Hansen, ed., *Studies on Graphs and Discrete Programming* (North-Holland, Amsterdam, 1981) 319–343.
[42] J. Wein and S.A. Zenios, Massively parallel auction algorithms for the assignment problem, *Proc. 3rd Symp. on the Frontiers of Massively Parallel Computation*, MD (Nov. 1990).