LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-438

# ON THE MASSIVELY PARALLEL SOLUTION OF THE ASSIGNMENT PROBLEM

Joel Wein
Stavros Zenios

December 1990

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

# On the Massively Parallel Solution of the Assignment Problem

Joel M. Wein[†]
Department of Mathematics, MIT
Cambridge, MA 02139 &
Thinking Machines Corporation
Cambridge, MA 02142-1214

Stavros Zenios [‡]
Department of Decision Sciences
The Wharton School
University of Pennsylvania
Philadelphia, PA 19104

*This paper will also appear as Thinking Machines Corporation Technical Report OP90-1.

[†]Much of this work was done while the author was a summer employee of Thinking Machines Corporation. Research also partially supported by an ARO graduate fellowship, Air Force Contract AFOSR-86-0078, DARPA Contract N00014-89-J-1988, and by NSF PYI Award CCR-89-96272, with matching funds from IBM, Sun Microsystems, and UPS.

[‡]Much of this work was done while the author was visiting Thinking Machines Corporation. Research also partially supported by NSF grants ECS-8718971 and CCR-8811135 and AFOSR grant 89-0145.

**Abstract**

In this paper we discuss the design, implementation and effectiveness of massively parallel algorithms for the solution of large-scale assignment problems. In particular, we study the auction algorithm of Bertsekas, an algorithm based on the method of multipliers of Hestenes and Powell, and an algorithm based on the alternating direction method of multipliers of Eckstein.

We discuss alternative approaches to the massively parallel implementation of the auction algorithm, including Jacobi, Gauss-Seidel and a hybrid scheme. The hybrid scheme, in particular, exploits two different levels of parallelism and an efficient way of communicating the data between them without the need to perform general router operations across the hypercube network. We then study the performance of massively parallel implementations of the two methods of multipliers. Implementations are carried out on the Connection Machine CM–2, and the algorithms are evaluated empirically with the solution of large scale problems. The hybrid scheme significantly outperforms all of the other methods and gives the best computational results to date for a massively parallel solution to this problem.

# 1 Introduction

With the advent of parallel computing the possibility has arisen of solving very large problems from several areas of scientific computing much more quickly than ever before. One area which is receiving increasing attention in this regard is the area of *network optimization*: specifically, problems of routing and allocating resources in a large network represented by a graph. There are a variety of large network optimization problems that would be desirable to solve in real time. One can imagine attempting to route traffic in a large city in real time or coordinating a large fleet of service people to respond to requests. Network models for air-traffic control have been under investigation since the early days of linear programming; nevertheless, complete global models for this very important problem are still beyond the solution capabilities of current technology. Models for planning under uncertainty in the financial markets also give rise to very large network-flow problems. Such problems are receiving recently renewed interest, since academics and practitioner believe that with parallel optimization methods such problems can be solved. A survey of large scale applications of network optimization are given in Dembo et al. [13], whereas Zenios [35] provides a survey of the status of parallel optimization methods.

In order to approach the ambitious tasks outlined above, we must understand our ability to produce efficient implementations of potential basic building blocks of such systems. The assignment problem is a key building block in several applications of network modeling. Given $n$ persons, $n$ objects, and a benefit $a_{ij}$ associated with the assignment of object $i$ to person $j$, the *assignment problem* is to find an *assignment* of each person to exactly one object that maximizes the total benefit. In graph theoretic terms, we wish to find the minimum weight perfect matching in a bipartite graph. The problem is recognized as a fundamental and practical problem in combinatorial optimization, with applications even in fields such as chemical engineering, as well as an important building block in the solution of a variety of more complex optimization problems, such as quadratic assignment problems [28], three dimensional assignment problems, traveling salesman problems [18], [25], and crew scheduling and vehicle routing problems [9].

The parallel computational complexity of the assignment problem is one of the most interesting open questions in the theory of parallel computation today. Mulmuley, Vazirani and Vazirani give a randomized algorithm with an $O(\log^2 n)$ worst case time bound in the PRAM model using a polynomial number of processors [29]. The number of processors is large and is only polynomial in the size of the input if the weights are input in unary. Therefore, at this point, this is a result primarily of theoretical importance. Nonrandomized sublinear time bounds have been achieved by several groups of researchers, including Goldberg, Plotkin, Shmoys and Tardos [21], and Goldberg, Plotkin and Vaidya [20]. These latter results lend credence to the possibility that massive parallelism may yield significant speedups for this problem.

In addition to these theoretical studies we have seen recently a surge of activities in the parallel implementation, and testing, of algorithms for the assignment problem. In particular, researchers have returned to some of the faster serial algorithms and re-examined them for their parallel potential. At the same time new algorithms have been developed, with a primary design requirement their suitability for parallel computing. In the latter class we put specifically the auction algorithm, initially introduced by Bertsekas [3] and further developed and extended in [4] and [7]. This algorithm has demonstrated good ability to take advantage of small scale parallelism, and is a suitable candidate for a method that can exploit the power of massively parallel, fine grain SIMD architectures.

A number of researchers have studied parallel solutions to large assignment problems on

smaller scale MIMD parallel machines. Kennington and Wang [27] developed a parallel version of the shortest augmenting path (SAP) code of Jonker and Volgenant [24], which they tested on the Symmetry S81, with up to ten Intel 80386 cpus. They report solutions to dense $1200 \times 1200$ assignment problems with cost range $[0 - 1000]$ in approximately 15 seconds and cost range $[0 - 10000]$ in an average of under 20 seconds. They also report that the auction algorithm did not achieve results comparable to the shortest augmenting path in a serial implementation, and hence it was not parallelized. Zaki [34] continued this study on an Alliant FX/8, parallelizing and vectorizing both algorithms. He confirmed the observations of Kennington and Wang, for certain problem categories. However, his results show that the auction algorithm achieves much better speedups than the SAP code and also vectorizes very well. As a result, the auction algorithm outperforms SAP by a large margin when implemented on a vector/parallel architecture. He reports solutions of $2000 \times 2000$ problems with cost range [0-10000] in approximately 30 seconds with the auction algorithm, and 2 minutes with SAP.

Kempka, Kennington and Zaki [26] tested the auction algorithm on the Alliant FX/8 without the $\epsilon$-scaling that typically makes the algorithm computationally effective and more stable. They report, nonetheless, solutions to a $1000 \times 1000$ dense problem with cost range $[0, 100]$ in under one second and a $4000 \times 4000$ problem in just over a half minute. Since they do not use scaling, their results are unpredictable: a $1000 \times 1000$ problem takes 12 seconds for cost range $[0, 1000]$, while a $2000 \times 2000$ problem with the same cost range takes over 255 seconds. For cost range 10000 they achieve an average of 33 seconds for $2000 \times 2000$ problems.

Balas, Miller, Pekny and Toth [2] implemented a parallel shortest augmenting path algorithm on the Butterfly Plus computer, with 14 processors. They were able to solve dense $1000 \times 1000$ problems with cost range $[0 - 1000]$ in an average of 9.39 seconds, and cost range $[0 - 10000]$ in 11.70 seconds. They also solved dense $2000 \times 2000$ problems with cost range $[0 - 10000]$ in 30 seconds, $3000 \times 3000$ in a minute, and a dense $30000 \times 30000$ problem with 900 million variables in less than an hour.

Bertsekas and Castanon [5] did an extensive study of several variants of the algorithm on 20% dense problems on the Encore Multimax. They tested both Jacobi and Gauss-Seidel versions and a block-Jacobi implementation. An interesting feature of this study is that they develop asynchronous, as well as synchronous, parallel implementations. The asynchronous algorithm has a substantial advantage over its synchronous counterpart. They were able to solve problems of size $1000 \times 1000$ in under 10 seconds.

Castanon, Smith and Wilson [10] studied the effectiveness of different synchronous implementations of the Gauss-Seidel auction algorithm and the shortest augmenting paths code of Jonker and Volgenant [24] for solving both dense and sparse assignment problems on a variety of architectures. They demonstrated speedups of up to 60 for the Gauss-Seidel implementation of the auction algorithm for problems of size $1000 \times 1000$.

The literature survey above indicates that the auction algorithm has proved competitive with other approaches to the assignment problem. It achieves moderate speedups on coarse-grain MIMD parallel implementations, and this gives the algorithm an additional advantage over competing methods that may not parallelize as well.

In this paper we describe the development of massively parallel implementations of algorithms for the assignment problem. We consider in particular: (1) the auction algorithm of Bertsekas, (2) an algorithm derived from the method of multipliers (MOM) due to Hestenes [22] and Powell [31] and (3) an algorithm developed as a specialization of the general algorithmic scheme of the

*alternating direction method of multipliers* (ADMOM) of Eckstein [14]. Eckstein has done an empirical study of this latter algorithm for general linear cost sparse networks on the Connection Machine CM-2. The computational results were encouraging for sparse assignment problems. Both methods of multipliers have tremendous potential for massive parallelism, and therefore are important to test on assignment problems.

All of our algorithms are implemented on the Connection Machine CM–2. Two traditional approaches to the parallel implementation of the auction algorithm, Jacobi and Gauss-Seidel, exploit only part of the power of the CM–2. We therefore introduce a hybrid algorithm that is able to exploit the parallelism of the machine at two different levels. This algorithm proves to be significantly faster than the other approaches. We then study the two different methods of multipliers.

The primary contributions of this paper are: the development of techniques for the massively parallel implementation of the auction algorithm on an SIMD architecture, the development of the parallel implementations of the two methods of multipliers for dense assignment problems, and the comparative study of the three algorithms. For the auction algorithm we develop three alternative methods for its parallelization, while the multipliers algorithms are implemented in what appears to be the most intuitive approach. The computational results indicate that the massively parallel implementation of the auction algorithm is very competitive with other, or the same, algorithms implemented on coarse-grain MIMD architectures. The comparisons with the methods of multipliers algorithms indicate that both are inferior to the auction algorithm for dense problems, despite their tremendous potential for massively parallel computing.

In this study we concentrate on the dense assignment problem, for several reasons. Most of the reports cited above also concentrate on the dense problem, and thus we can use the results of this paper as benchmarks against the work reported by others. Furthermore, we expect that the insight gained from the dense implementation on the performance of the algorithm will carry over to sparse problems. Finally, the dense problem yields a very communication-efficient representation on the Connection Machine. The data structures that would be necessary for the sparse implementation of assignment problems on the CM have been reported on in [36]. Experience with the dense and sparse implementation of nonlinear network optimization problems in Nielsen and Zenios [39] indicate that the sparse implementation can outperform significantly the dense implementation for problems that are fairly sparse, even if it does not use well structured communication patterns like the dense implementation.

While this is the first — to the best of our knowledge — study of *massively* parallel algorithms for the assignment problem, there has been a variety of recent related work on massively parallel network optimization. Goldberg implemented a fast maximum flow algorithm [19]. Zenios and Lasken [38] solved nonlinear network problems; Zenios [37] developed algorithms for multi-commodity transportation problems; Nielsen and Zenios [40] developed algorithms for stochastic network optimization models arising in financial applications. Wein [32] studied massively parallel algorithms for minimum-cost circulation. Eckstein [14], [15] has extensively studied the alternating step method for transportation problems. As stated above, his results for linear cost networks are not encouraging, but the results for sparse quadratic transportation problems are quite good and are competitive with the massively parallel row-action algorithm of Zenios and Censor [11]. Furthermore, his method appears to be the able to solve problems with mixed linear and quadratic objective terms with little additional difficulty.

The rest of this paper is organized as follows. In section 2 we discuss the algorithms for which we developed massively parallel implementations. In section 3 we discuss the implementations

of these algorithms on the Connection Machine CM–2. Section 4 contains computational results for these implementations, and section 5 contains our conclusions.

# 2 Algorithms for the Assignment Problem

We define the problem formally as follows: Find an assignment $A$ of objects to people ($A(i) = j$ means that object $i$ is assigned to person $j$) so that $\sum_{A(i)=j} a_{ij}$ is maximized. In a globally optimal solution any given person may not be assigned to his most valuable object. However, for a globally optimal assignment it is possible to assign a price $\pi_i$ to each object $i$, so that if each person $j$ views the profit associated with object $i$ as $a_{ij} - \pi_i$ then each person is assigned to his most *profitable* object. This fact can be understood as a consequence of linear programming duality.

## 2.1 The Auction Algorithm

The auction algorithm finds the optimum assignment by finding such prices for all the objects. It produces an assignment $A$ and prices $\pi_i$ such that

$$a_{iA(i)} - \pi_{(i)} \geq \max_{j=1,\ldots,n} (a_{jA(i)} - \pi_j).$$

The algorithm starts with each object assigned an arbitrary price; prices are adjusted upwards as people bid for their most profitable object. Each iteration of the algorithm consists of one or several currently unassigned people choosing the object that is most profitable to them and submitting a "bid" on the object. Each object that has been bid upon is assigned to the highest bidder, adjusts its price to the bid, and deassigns the person to whom it was previously assigned (if anyone).

Formally, each person bidding calculates the profits $p_{best}$ and $p_{next}$ associated with his two most profitable objects, and then bids $\pi_i + p_{best} - p_{next}$ on his best object $i$. A bid-upon object then is assigned to the highest bidder and sets its price to that bid.

### 2.1.1 Epsilon Scaling

Epsilon-scaling is used in the auction algorithm in order to improve upon the worst-case time bounds and computational behavior. We relax the condition that an object bids upon and is assigned to its favorite object:

$$a_{iA(i)} - \pi_i \geq \max_{j=1,\ldots,n} (a_{jA(i)} - \pi_j).$$

Instead we merely require that

$$a_{iA(i)} - \pi_i \geq \max_{j=1,\ldots,n} (a_{jA(i)} - \pi_j) - \epsilon.$$

Such an assignment is called *$\epsilon$-optimal.* The algorithm runs in a series of *phases*, each phase taking an $\epsilon$-optimal assignment and returning a set of prices and an assignment that is $\epsilon' = (\epsilon/c)$-optimal, for some user-specified constant $c$. Bertsekas proved that if an assignment is $\epsilon$-optimal

with $\epsilon < \frac{1}{n}$, and the $a_{ij}$ are all integers, then the assignment is globally optimal [3]. Define $M_a = \max_{ij} a_{ij}$. Since *any* assignment is $M_a$-optimal, we can start with $\epsilon = M_a$ and in $\log(M_a n)$ phases we will produce an assignment that is $(\frac{1}{n+1})$-optimal and thus globally optimal. Each phase of the algorithm is a mini-auction as described in the previous section, except that instead of bidding $p_{best} - p_{next}$, a person can bid $p_{best} - p_{next} + \epsilon$. Each phase produces a $\epsilon$-optimal assignment, and by successively lowering $\epsilon$ in this fashion we obtain an optimal assignment. The worst-case sequential complexity of the algorithm is $O(n \times M_a \times \log(M_a n))$. A summary of the algorithm is as follows.

**Step 0** (Initialization) $\epsilon \leftarrow \max_{ij} a_{ij}$, $\pi_j \leftarrow 0$.

**Step 1** Auction:

**1.1** Some subset of the set of unassigned people determine and bid on their favorite object.

**1.2** Each bid-upon object determines its highest bidder, raises its price to that bid, and assigns itself to that person, deassigning the person to whom it was previously assigned (if anyone).

**1.3** If any person is unassigned goto 1.1.

**Step 2** If $\epsilon < \frac{1}{n}$ **Stop**; else $\epsilon \leftarrow \epsilon/2$. People whose assignments are no longer $\epsilon$-optimal deassign themselves.

**Step 3** Goto Step 1.

### 2.1.2 Jacobi vs. Gauss-Seidel

Note that in the most general form of the algorithm any subset of those people unassigned can bid simultaneously. The two traditional parallel variants of the auction algorithm are the Jacobi version and the Gauss-Seidel version. By the Jacobi version we refer to an algorithm in which *all* unassigned people bid simultaneously on their favorite objects before the prices are adjusted, whereas in Gauss-Seidel only one person bids at a time. Since in the Gauss-Seidel version each bid takes advantage of the updated price information of all the previous bids, it usually takes fewer total bids to produce an optimal assignment. The Jacobi method, however, has greater potential for a massively parallel implementation.

In general the terms "Jacobi" and "Gauss-Seidel" are not used solely with regard to the auction algorithm. "Jacobi" is used to refer to a method where the prices (dual variables) at time $t+1$ are updated only with respect to the information at time $t$, whereas a "Gauss-Seidel" iteration updates a dual variable with respect to the most recent information. Thus a Jacobi method allows the updating of all prices in parallel, whereas a Gauss-Seidel method allows two prices to be updated in parallel only if the update of one does not depend on the relevant values of the other. This Jacobi/Gauss-Seidel distinction is one of the primary differences between the two methods of multipliers we discuss.

## 2.2 The Method of Multipliers Algorithm

The *method of multipliers* (MOM) is a general method for a variety of problems in convex programming [22], [31]; we summarize here a specialization of the method to assignment problems

suggested in [6] and we refer the reader there for a full development and explanation of the algorithm. Since the methods of multipliers we describe here are specializations of multiplier methods for linear programs, we first formulate the assignment problem as a linear program, and without loss of generality as a minimization problem. We let $E$ denote the set of edges of the network; in our dense case $E$ contains an edge $(i,j)$ between every person $i$ and object $j$.

$$\text{minimize} \sum_{\{(i,j)\in E\}} a_{ij} f_{ij}$$

$$\text{subject to}$$

$$\sum_{\{j|(i,j)\in E\}} f_{ij} = 1 \ \forall i = 1, \cdots n,$$

$$\sum_{\{i|(i,j)\in E\}} f_{ij} = 1 \ \forall j = 1, \cdots n,$$

$$0 \leq f_{ij} \leq 1 \quad \forall (i,j) \in A.$$

We assign dual prices $r_i$ and $p_j$ to the equality constraints. The method of multipliers for this problem results in a Gauss-Seidel iteration to minimize the Augmented Lagrangian. It has the form:

**Step 0** $f_{ij} \leftarrow 0, r_i \leftarrow 0, p_j \leftarrow 0.$

**Step 1** Update the values of $f$ as follows

$$f_{ij} = \left[ f_{ij} + \frac{1}{2c(t)} [r_i(t) + p_j(t) - a_{ij} + c(t)(y_i(t) + w_j(t))] \right]^+, \forall \text{edges}(i,j)$$

where $y_i(t)$ and $w_j(t)$ are given in terms of $f_{ij}(t)$ by

$$y_i(t) = (1 - \sum_{\{j|(i,j)\in A\}} f_{ij}(t)), \forall i = 1, \cdots n$$

$$w_j(t) = (1 - \sum_{\{i|(i,j)\in A\}} f_{ij}(t)), \forall j = 1, \cdots n,$$

$c(t)$ is a nondecreasing sequence of of positive constants and $[x]^+$ indicates the projection of $x$ onto $[0,1]$.

**Step 2** At the end of the minimization yielding $f_{ij}(t+1)$, $y_i(t+1)$ and $w_j(t+1)$ update the prices $r_i$ and $p_j$ according to

$$r_i(t+1) = r_i(t) + cy_i(t+1), \forall i = 1, \cdots, n,$$

$$p_j(t+1) = p_j(t) + cw_j(t+1), \forall j = 1, \cdots, n$$

**Step 3** Check for convergence; if iteration has not converged goto Step 1.

### 2.3 The Alternating Direction Method of Multipliers

The *alternating direction method of multipliers*(ADMOM), due to Eckstein [14], takes a Jacobi approach to updating the augmented Lagrangian This method as well has a variety of applications to convex programming, [6], [14], [15], [16], [17]. The application of this method to the assignment problem results in a Jacobi-type algorithm which is more suitable for massively parallel computation than the method of multipliers. The iteration proceeds as follows.

**Step 0** $f_{ij} \leftarrow 0, r_i \leftarrow 0, p_j \leftarrow 0.$

**Step 1** Update the $f_{ij}$ as follows

$$f_{ij}(t+1) = \left[f_{ij}(t) + \frac{1}{2c}[r_i(t) + p_j(t) - a_{ij} + c(y_i(t) + w_j(t))]\right]^+ \forall \text{edges}(i,j),$$

$$r_i(t+1) = r_i(t) + cy_i(t+1), \forall i = 1, \cdots, n,$$

$$p_j(t+1) = p_j(t) + cw_j(t+1), \forall j = 1, \cdots, n$$

where $y_i(t)$ and $w_j(t)$ are given in terms of $f_{ij}(t)$ by

$$y_i(t) = \frac{1}{n}(1 - \sum_{\{j|(i,j)\in E\}} f_{ij}(t)), \forall i = 1, \cdots n,$$
$$w_j(t) = \frac{1}{n}(1 - \sum_{\{i|(i,j)\in E\}} f_{ij}(t)), \forall j = 1, \cdots n,$$

and $c$ is a constant.

**Step 2** Check for convergence; if iteration has not converged goto Step 1.

Note that in ADMOM all the $f_{ij}$ can be updated simultaneously while MOM relies on $y$ and $w$ being up to date, and therefore only one value $f_{ij}$ can be updated in each row and column at a time. Therefore we can only do $n$ updates at a time for MOM while we can do $n^2$ for ADMOM; however, due to the Gauss-Seidel nature of the minimizing iteration in MOM, we would expect that the algorithm would converge much more quickly. Furthermore the price updates for ADMOM are divided by the number of arcs incident to that node, which for dense problems is $n$. Therefore for large dense problems the prices will only change a small amount in each iteration and therefore the number of iterations has the potential to be large.

## 3 Designs for Massively Parallel Implementation

In this section we discuss the issues involved in implementing efficiently the algorithms of the previous section on a Connection Machine CM–2.

## 3.1 The Connection Machine CM–2

The Connection Machine CM–2 is a massively parallel computer with up to 65,536 processors. Each processor has a single-bit processing unit and 64K or 256K bits of local RAM. The processors run in SIMD mode and are connected in an $n$-cube topology. The system software provides global maximum operations as well as *scan* and *spread* operations that are parallel prefix operations [8]. The CM–2 uses a front end such as a SUN-4, VAX or Symbolics Lisp Machine. Parallel extensions to the programming languages LISP, C and FORTRAN, via the front-end, allow the user to program the Connection Machine and the front-end system. For further information see [12] and [23].

A Connection Machine can emulate a large number of processors by having each physical processor simulate a number of *virtual* processors. The ratio of the number of virtual processors to the number of physical processors is referred to as the *virtual processor ratio*, or *vp-ratio*. Using standard Gray coding the processors of the CM can be configured as a $k$-dimensional grid; to represent a $n \times n$ assignment problem we configure the CM as a $N \times N$ grid, where $N$ is $n$ rounded up to the nearest power of two. Row $i$ is associated with object $i$ and column $j$ is associated with object $j$. In particular, processor $(i, j)$ stores the value $a_{ij}$ of object $i$ to person $j$, local variables applicable to person $j$ such as the most profitable object to that person, and local variables applicable to object $i$, such as its price. A specified number of virtual processors, along with a configuration of the machine, is called a *vp-set*, and it is possible to use several different vp-sets in the same computation and to switch between them when desired.

A mapping of virtual processors in a grid to the physical processors of the CM is known as a *geometry*. The user has the freedom to dictate a variety of the features of this mapping. In particular we exploit the ability to specify which axes or directions of the grid should have more physical processors representing them, and which should have more virtual processors. In other words, we can specify that virtual processors that are adjacent along one axis are on different physical processors, and that processors that are adjacent on another axis are on the same physical processor. The mapping of the $N \times N$ grid onto the physical CM will differ among our implementations, and we will use alternate representations as well, but this is the basic representation common to all the codes.

## 3.2 Massively Parallel Implementations of the Auction Algorithm

In this section we discuss the implementation of the Jacobi and Gauss-Seidel variants of the auction algorithm, together with a new version that we refer to as a "hybrid" algorithm.

### 3.2.1 The Jacobi Algorithm

In the implementation of the Jacobi[1] algorithm we have one $N \times N$ vp-set which is used both to store the data and for computation. It is mapped onto the physical CM processors in the default fashion, which balances the two axes so that they have a comparable physical processor/virtual processor makeup. A detailed summary of the Jacobi implementation is as follows.

[1] Preliminary experiences with this algorithm have already been reported on in [30]; it is included here for completeness and in order to present additional computational results. The description of the algorithm is adapted from that paper.

| Number of Bidders | Fraction of Iterations |
|---|---|
| 1 | 38.60% |
| 1-10 | 82.35% |
| 1-100 | 95.95% |
| 1-500 | 98.78% |

Table 1: Statistics on Number of Bidders active

**Step 0** We keep a copy of $\epsilon$ in each processor. Using a global maximum operation and a broadcast, set $\epsilon \leftarrow (n+1)M_a$. In each processor set $\pi \leftarrow 0$. We keep two boolean variables in each processor: `assigned-here`, which is True in processor $(i,j)$ if object $i$ is assigned to person $j$, and `person-assigned`, which is True in all processors of column $j$ if person $j$ is assigned to an object. In all processors set `person-assigned` $\leftarrow$ False and `assigned-here` $\leftarrow$ False. Also scale all values of $a$ by $n+1$.

**Step 1** Determine if everyone is assigned (a global or operation of `person-assigned`). If a person is unassigned proceed to Step 2. If every person is assigned to an object and $\epsilon < 1$ the algorithm terminates. If $\epsilon > 1$ reduce its value, deassign everyone whose current assignment is no longer $\epsilon$-optimal for the new $\epsilon$, and proceed to Step 2.

**Step 2** Select all the processors associated with unassigned people. Set `profit` $\leftarrow a - \pi$. Within each column $j$ find the row index $i_j$ of the most profitable object by forming the concatenation of the profit and column number $j$ in each processor and doing a grid spread-with-max. Set `best` $\leftarrow i_j$. Turning off processor (`best`, $j$) do another grid spread-with-max to find the profit $p$ of the next-best object and set `next-best` $\leftarrow p$. Person $j$ bids on object `best` by setting the variable `bid` in processor (`best`, $j$). The value of the bid is computed as follows: Let $w_{\texttt{best},j}$ be the maximum profit from all objects except `best`. The bid from $j$ to `best` is $a_{\texttt{best},j} - w_{\texttt{best},j} + \epsilon$.

**Step 3** Using a grid spread-with-max along the rows, determine the maximum bid on each object and update the prices $\pi$ within the columns. For all objects bid upon, assign the object to the highest bidder by setting `assigned-here`. Update `person-assigned` using or-grid-scans within the columns. Go to Step 1.

The CM has the ability to potentially perform thousands of bids at once; therefore, this seems to be a very attractive method for this architecture. However, this approach leads to a large sequential tail. Most people are assigned to objects very quickly, and the bulk of the computational time is spent in establishing the last few assignments, hence greatly reducing the amount of parallelism. Table 1 gives a typical breakdown of processor utilization encountered in a $1000 \times 1000$ problem with cost range $[0 - 1000]$.

Two optimizations were implemented to minimize this tail effect. The first was to optimize the case when only one bidder was active: the preferred object for that sole bidder was found by a global maximum over all the processors as opposed to the more powerful but slower max-scan operation. The latter was unnecessary for the case of one bidder.

The second optimization was the truncation of the tail: instead of running each phase until a complete $\epsilon$-optimal assignment was achieved, the auction was terminated when $k\%$ of the people

| Problem Size | Maximum Cost | Speedup(its.) | Speedup(Time) |
|---|---|---|---|
| 128 | 100 | 5.62 | 4.61 |
| 128 | 1000 | 4.30 | 3.70 |
| 256 | 100 | 3.23 | 3.28 |
| 256 | 1000 | 7.37 | 6.35 |

Table 2: Improvements in iterations and time gained by truncation of the tail. Each entry is an average over five randomly generated examples.

have been matched. $k$ increases as *epsilon* decreases, so that $k = 100$ in the last phase. This optimization resulted in substantial speedups. An implementation that initially matches only 80%, and matches progressively more in successive phases does an average of 5.1 times fewer iterations than an implementation that completes each phase, on randomly generated problems of size 128-256. (See Table 2.)

### 3.2.2 The Gauss-Seidel Algorithm

In this version, where we do one bid at a time, we must continue to store the $n \times n$ problem on the CM, which will necessarily be configured at a high vp-ratio. We would like, however, to perform the computations at a lower vp-ratio, and therefore more quickly, on a grid of reduced dimension. We introduce a mapping of the $N \times N$ grid to the physical machine that will enable us to extract the information necessary for one bid, and compute the bid in a vp-set with vp-ratio 1. We do this in a fashion that requires *no communication* between the two vp-sets.

We define the geometry of the $N \times N$ vp-set, which we will call **data-vp-set**, so that the $y$ axis is physical. Processors $(i, j)$ and $(i, k)$ are on different physical processors for all $j \neq k$. All of the virtual bits are along the $x$ axis; processors $(0, j), (1, j), \ldots, (\text{vp-ratio} - 1, j)$ will all be mapped to the same physical processor, as will all processors $(q, j)$, where $q \in [\text{vp-ratio} * k, \text{vp-ratio} * (k + 1) - 1]$. We define a second vp-set, **compute-vp-set** , of vp-ratio 1 that has $N$ rows and (Number of Physical CM processors)$/N$ columns. For example, if $N = 1024$ and we are working on a Connection Machine with 16,384 processors, **compute-vp-set** will be a $1024 \times 16$ vp-set (1024 rows, 16 columns). A processor in **compute-vp-set** shares the same physical processor with 64 processors in the **data-vp-set**. Note that an entire column in **data-vp-set** shares the same physical processors with one of the columns of **compute-vp-set**. We will transfer the information we need for one bid from **data-vp-set** to **compute-vp-set** by having a processor in **compute-vp-set** just point to the relevant data in **data-vp-set**. See figure 1.

To execute the bid for the $i$th person we select his column $c_{data}$ in **data-vp-set**, and identify the column in **compute-vp-set** with which it is coincident: $c_{compute} = \lfloor \frac{c_{data}}{vp-ratio} \rfloor$ . Each parallel variable has a pointer to the physical location where its data resides; we simply change the pointer of the parallel variable in **compute-vp-set** to point at the data in the **data-vp-set**. For example, in our $1024 \times 1024$ problem on a 16,384 processor machine, suppose person 303 is bidding. All the relevant information resides on the same physical processors as does column 4 of the compute vp-set. Pointers are changed so that column 4 points to the data of column 303 in **data-vp-set**, and the bid is carried out. Note that the price information need only be stored in **compute-vp-set**. The Gauss-Seidel algorithm is thus as follows:

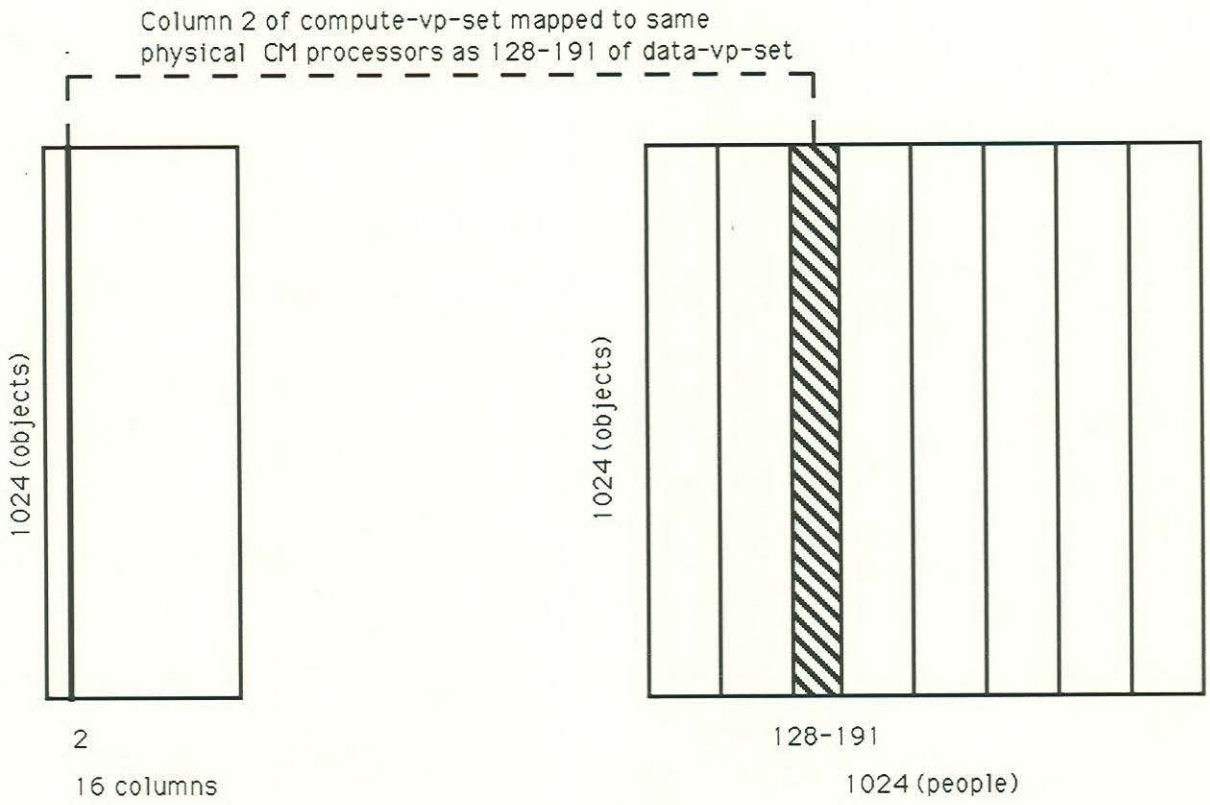


Figure 1: The two vp-sets for the Gauss-Seidel algorithm, **compute-vp-set** and **data-vp-set** for a $1024 \times 1024$ problem on a 16K CM. Each vertical block of **data-vp-set** represents 64 virtual columns that all reside on the same physical column. Only 8 blocks are portrayed here.

**Step 1** Pick an unassigned person and calculate with which physical column he is associated; set the appropriate pvars in **compute-vp-set** to point to his values. (Need to change only one pointer per pvar.)

**Step 2** In **compute-vp-set**, calculate his bid. (This requires two global maximum computations: one to calculate the best object and one for the next best.)

**Step 3** Update the price information, which only need be kept in **compute-vp-set**.

**Step 4** Goto step 1.

In contrast to the Jacobi implementation, where all information is kept on the CM, it is more efficient here to keep track of who is assigned, and to what object, in front-end lists and arrays. This avoids significant amounts of communication. In fact we utilize a copy of the $a_{ij}$ that is kept on the front end as well to calculate the bid and in this way avoid CM to front-end communication time.

Note that from the end of one phase to the start of the next, when epsilon is decreased to $\epsilon'$, often many of the $\epsilon$-optimal assignments from the previous phase are $\epsilon'$-optimal as well, and need not be recomputed in the next phase. In the Jacobi code these assignments are identified and preserved for the next phase; this is easy with a Jacobi representation since each assignment can be checked in parallel. In the Gauss-Seidel case, when this information is stored on the front end, it is a sequential computation to determine who is $\epsilon'$-optimal and thus we do not check. Not preserving these assignments does increase the total number of Gauss-Seidel bids, since we are throwing away information, but the gain in computation time outweighed the increase in iterations; thus we chose not to preserve them. We tested our Gauss-Seidel implementation against a sequential Gauss-Seidel implementation that we obtained from D.P. Bertsekas on problems of size $64 - 256$ and found that the number of bids they performed was comparable.

### 3.2.3 The Hybrid Jacobi/Gauss-Seidel Algorithm

The Jacobi code is very efficient when a large number of people are unassigned and are bidding and the Gauss-Seidel code makes effective use of the machine when there are few active bidders. It can do one bid at a time very quickly and also decreases the total number of bids needed. These characterizations suggest that a better use of the CM is a hybrid combination of these approaches. Execute the Jacobi algorithm early in the phase when large numbers of people are unassigned. When most are assigned, switch to Gauss-Seidel. Note that several theoretical results have taken similar approaches in devising algorithms for the assignment problem with improved worst case running times [1], [20].

Making the transition from Jacobi to Gauss-Seidel will require a certain amount of communication of the data during a phase, since the way that the grids are mapped onto the Connection Machine is different for Jacobi and Gauss-Seidel. This cost is far outweighed by the gains in computation time.

The hybrid algorithm uses two thresholds, $thresh1$ and $thresh2$. $thresh1$ determines in which phases we employ both methods; $thresh2$ determines when in the phase we switch to Gauss-Seidel. If a phase is aiming to assign $k\%$, $k\% > thresh1$ then we switch to Gauss-Seidel when $thresh2 \times k\%$ of the people are assigned. Computational testing showed that for problems of size $1000 \times 1000$ and $2000 \times 2000$ the best setting of these parameters, although different for

| Size | cost | Jacobi Bids | Jacobi Iterations | Average Parallelism | GS Bids |
|---|---|---|---|---|---|
| 128 | 100 | 3837 | 147 | 26.49 | 1567 |
| 128 | 1000 | 4114 | 215 | 20.77 | 2099 |
| 256 | 100 | 10210 | 813 | 14.56 | 5102 |
| 256 | 1000 | 9054 | 264 | 40.56 | 6244 |

Table 3: Jacobi vs. Gauss-Seidel Statistics

different cost ranges, was $thresh1 = thresh2 = 97$ or 98. Interestingly, for both problems this is close to the $\sqrt{n}$ turning point used theoretically in [1].

### 3.3 Massively Parallel Implementations of the Methods of Multipliers

In contrast to the auction algorithm, which only has the *potential* to do $n$ bids at a time, MOM *always* does $n$ updates at a time and ADMOM does $n^2$. Therefore these methods are very attractive for parallel computation. The description of the Connection Machine implementations of these algorithms is relatively straightforward given our previous discussion. We configure the Connection Machine as an $N \times N$ geometry, with equal preference given to the physical makeup of each axis, (as we did with the pure Jacobi auction algorithm). Each processor $(i, j)$ stores the current value of $f_{ij}$, $p_i$, $r_j$, $y_i$, and $w_j$. For the ADMOM algorithm, we need merely to do one spread-with-add operation along each axis in order to calculate $\sum_{\{j|(i,j)\in E\}} f_{ij}(t), \forall i = 1, \ldots, n$, and $\sum_{\{i|(i,j)\in E\}} f_{ij}(t), \forall j = 1, \ldots, n$; then all that is required is several arithmetic operations that all happen within each processor with no further communication required. For the MOM algorithm we must select groups of $n$ processors such that only one processor is selected in each row and column. The strategy that we use is to select processors $(i, j)$ such that $i+j \equiv k (\mathrm{mod}\ n)$, and loop over $k = 0, \ldots, n-1$.

## 4 Computational Results and Discussion

In this section we discuss the performance of the two algorithms on dense problems. All the problem data was generated by the Connection Machine random number generator.

### 4.1 The Performance of the Auction Algorithm

The auction algorithms were initially implemented in a combination of *Lisp and Lisp/Paris and were run with a SUN4 front end. A comparison of the number of bids done by the Jacobi and Gauss-Seidel codes on problems of moderate size is given in table 3. We see that the Gauss-Seidel code can do significantly fewer bids than Jacobi, by as much as a factor of two or greater. We also see that, as expected, the average number of people bidding in the Jacobi code, is much less than $n$.

Table 4 gives data on the running times of the algorithms on fully dense problems of size $n = 1024$, on a 16,384 processor CM2. Each running time is given in seconds and is the average of five problems. The Hybrid algorithm is faster than both the Jacobi and Gauss-Seidel codes

| Size | cost | Jacobi Time | Gauss-Seidel Time | Hybrid Time |
|---|---|---|---|---|
| 1024 | 100 | 197.5/106.2 | 585.5/124.8 | 105.1/25.5 |
| 1024 | 1000 | 414.2/219.5 | 242.7/46.5 | 58.6/22.1 |
| 1024 | 10000 | 276.7/146.0 | 222.2/44.3 | 92.3/20.8 |

Table 4: Running times of the algorithms, in seconds, averaged over five random examples. Both total and CM time are given.

in all cost ranges; this was true as well for a number of smaller problem sizes. Further, this also held true for a variety of settings of the parameters of the code, such as factor by which we divide epsilon in each iteration, the initial percentage matched, etc. This led us to believe that the superiority of the hybrid approach was fairly robust with respect to modest modifications to the algorithms.

Note that the total times are significantly larger than the CM times. This partially reflects the strategy of letting the front end execute as much of the inherently sequential part of the problem as possible, and partially reflects the slowness of the Lisp front end code.

Based on this testing we chose the hybrid algorithm as the most successful and recoded it in C/Paris. The C front end code runs much faster, and this led to significant improvements in the overall running times; the discrepancy between front end times and CM time became very small. In tables 5 and 6 we give results on the performance of the algorithms on both 16K and 32K machines, for problems of size 1000-2000, over various cost ranges. Both the total number of iterations and the number of Jacobi iterations are reported. Each number is the average of ten randomly generated examples; we used different sets of examples for the 16K and 32K machine in order to give some idea of the variation possible in algorithm performance over two similar sets of problems of the same size and cost range.

The number of Jacobi iterations is surprisingly small, but during these iterations hundreds of bids are carried out at once. Another interesting fact is that for a specific problem size and cost range the number of Jacobi iterations is always about the same; for almost all size and cost ranges it was never more than five away from the average.

We note that the cost range 100 for size 1000 problems is particularly difficult for our implementation; this is reflected both in the increased number of iterations and in the fact that on approximately 25% of the random instances we generated the code ran for more than 100,000 iterations, although it always terminated. After initial testing we ran each code only up to 100000 iterations and thus those examples of size 1000 and cost 100 that ran longer are not included in the averages.

We also note that given our vp-ratio 1 implementation of a Gauss Seidel bid, the time for one bid is not dependent on problem size or machine size as long as $n$ is smaller than the size of the machine. Our code averages 1000 Gauss-Seidel bids per second on both a 16K and 32K CM–2. This of course is not the case for the Jacobi phase, e.g. one Jacobi parallel bid for a $1000 \times 1000$ problem on a 16K CM–2 takes .07 seconds, whereas on a 32K machine one parallel bid takes .04 seconds.

| Size | cost | Time | Total Iterations | Jacobi Iterations |
|---|---|---|---|---|
| 1000 | 100 | 34.6 | 16252 | 128 |
| 1000 | 1000 | 22.2 | 13637 | 119 |
| 1000 | 10000 | 15.4 | 5031 | 141 |
| 1000 | 100000 | 15.3 | 4923 | 144 |

| Size | cost | Time | Total Iterations | Jacobi Iterations |
|---|---|---|---|---|
| 1000 | 100 | 28.7 | 21850 | 126 |
| 1000 | 1000 | 17.5 | 10141 | 120 |
| 1000 | 10000 | 8.1 | 2000 | 141 |
| 1000 | 100000 | 9.8 | 3578 | 144 |

Table 5: Running times of the C/Paris hybrid auction algorithm on $1000 \times 1000$ problems. The top table is for a 16K machine, the bottom table for a 32K machine: Time is in seconds, averaged over ten random examples. 15 of the 20 examples with cost range $[1 - 100]$ tested ran for more than 100000 iterations, and their running times are not included.

| Size | cost | Time | Total Iterations | Jacobi Iterations |
|---|---|---|---|---|
| 2000 | 100 | 60.2 | 2817 | 242 |
| 2000 | 1000 | 96.3 | 60936 | 148 |
| 2000 | 10000 | 60.4 | 23657 | 152 |
| 2000 | 100000 | 57.9 | 18004 | 156 |

| Size | cost | Time | Total Iterations | Jacobi Iterations |
|---|---|---|---|---|
| 2000 | 100 | 33.5 | 2368 | 241 |
| 2000 | 1000 | 50.3 | 22955 | 150 |
| 2000 | 10000 | 45.3 | 26739 | 151 |
| 2000 | 100000 | 36.3 | 15263 | 157 |

Table 6: Running times of the C/Paris hybrid auction algorithm on $2000 \times 2000$ problems. The top table is for a 16K machine, the bottom table for a 32K machine. Time is in seconds, averaged over ten random examples.

| Cost | c=1 | c=5 | c=10 | c=25 | c=100 | c=500 |
|---|---|---|---|---|---|---|
| 100 | (1000,2) | (400,2) | (400,2) | (550,2) | (1400,2) | (5150,2) |
| 1000 | (8130,3) | (2880,10) | (1540,10) | (690,10) | (620,10) | (3450,8) |
| 10000 | * | * | (9800,1) | (6040,10) | (1740,10) | (730,10) |
| 100000 | X | X | X | X | (7750,10) | (2720,10) |

Table 7: Performance of ADMOM on dense problems of size 64. each combination of cost and $c$ was tested on 10 randomly generated problems. The entry $(x, y)$ = average of $x$ iterations on the $y/10$ problems that converged within 10000 iterations. A $*$ indicates that none of the tested examples in that category converged within the limit, and an $X$ indicates that that category was not tested fully since preliminary testing indicated it was sure not to converge.

### 4.2 The Performance of the Multiplier Methods

We began by testing the methods of multipliers on small problems in order to understand their behavior. The performance of MOM and ADMOM on randomly generated problems of size $64 \times 64$ and $256 \times 256$ is recorded in tables 7, 8, and 9. In table 8 the entry $(x, y)$ means that ADMOM ran for an average of $x$ iterations before converging, and that it converged within 10000 iterations for $y/10$ examples. We imposed an arbitrary cutoff here of $10,000$ iterations. An asterisk indicates that the methods never converged with that parameter setting on that cost range; an X means that we did not test that combination fully since initial testing indicated it would not converge. Based on our experience with ADMOM and some preliminary testing of MOM we chose a reduced set of $c$ on which to carefully test MOM. Table 8 gives the results; here we imposed an larger arbitrary cutoff of 20000 iterations, since due to the Gauss-Seidel nature of the algorithm the number of iterations is expected to be larger. Ten random examples were considered for each cost range; again, * should be interpreted as meaning none of the random examples converged.

The tests on the size 64 problems indicate that indeed the number of minimizations of the augmented lagrangian is much smaller for MOM than for ADMOM, but the tests on size 256 problems, given in table 9, indicate that total number of iterations of MOM gets unmanageable for size 256 problems, since one minimization requires 256 iterations of the inner loop. A further feature of MOM is that fairly frequently it returns non-integral but very close to optimal solutions; this behavior was not observed with ADMOM. The number (out of 10) of solutions that were integral is recorded in the table 8 as well.

Both methods were highly sensitive to choice of $c$, and different values of $c$ were preferable for different cost ranges.

Given this preliminary data we deemed it unnecessary to test MOM on $1000 \times 1000$ problems, due to its disappointing behavior on smaller ones. We did test ADMOM on dense problems of size $1000 \times 1000$, for $c$ equal to each of $25, 100, 500, 1000$. We tested all 4 cost ranges with each $c$, on five randomly generated examples. None of the examples converged within 10000 iterations, which was approximately 2 minutes of time on a 32K CM–2. We thus conclude that for problems of this size MOM and ADMOM are inferior to a hybrid auction approach and in general not practical ways of solving large dense problems.

| Cost | c=25 | # Integral | c=100 | # Integral | c=500 | # Integral |
|---|---|---|---|---|---|---|
| 100 | (9850,10) | 0 | (18430,1) | 0 | (13300,9) | 0 |
| 1000 | (5766,10) | 7 | (11460,9) | 5 | (16410,10) | 1 |
| 10000 | (7360,10) | 10 | (3827,10) | 9 | (5887,10) | 4 |
| 100000 | * | 0 | (12460,9) | 9 | (4530,10) | 8 |

Table 8: Performance of MOM on dense problems of size 64 of various cost ranges.

| Cost | c=25 | c=25 | c=100 | c=100 | c=500 | c=500 |
|---|---|---|---|---|---|---|
| 1000 | (26.5,2) | * | (58.5, 2) | * | (196,1) | * |
| 10000 | (86,7) | (181.7,2) | (39.9,7) | * | (67.8,8) | * |
| 100000 | * | * | * | * | (57.7,10) | * |

Table 9: Data on $256 \times 256$ problems. We imposed an arbitrary cutoff of 20000 iterations. First column for each value is ADMOM, second is MOM.

### 4.3 Comparisons With Other Codes

In table 10 we compare our computational results with those reported by [2], [26] and [27]. Our algorithm seems to be comparable or superior at cost ranges where the maximum cost is at least 10 times the problem size, and compares particularly poorly in small cost ranges. To the best of our knowledge most of the computational studies on the auction algorithm started from a well-developed and tested code authored by Dimitri Bertsekas and Paul Tseng, and thus incorporates their experience in testing this algorithm. Due to the different architecture on which we were working it was not possible to directly adapt this code, and we were interested largely in the issues involved in creating an efficient massively parallel implementation of this algorithm. We believe that some of the heuristics developed by Bertsekas and Tseng could be incorporated into our code in modified form and potentially significantly improve the number of iterations required.

## 5 Conclusions

We have seen that for large dense problems the two methods of multipliers take good advantage of the massive parallelism of the Connection Machine, but are not computationally effective methods for dense problems due to the large number of iterations required for convergence. The auction algorithm is more difficult to implement efficiently on the Connection Machine, but we have presented several methods to achieve an implementation that is competitive with the best MIMD algorithms on large problems.

Our results raise the question of whether other currently known algorithms might perform even better in a massively parallel environment. One class of algorithms we have not implemented is that of shortest augmenting paths. The success of such an approach would depend heavily on the quality of the shortest path code utilized, however, and developing a sufficiently fast shortest paths algorithm seems difficult. A parallelized Bellman-Ford approach is subject to the same tail problems that we experience with network algorithms: most of the paths are found quickly,

| Cost | Hybrid | Balas et. al | Kempka Kennington Zaki | Zaki | Kennington Wang |
|---|---|---|---|---|---|
| 100 | 28.7 | 2.01 | .758 | .56 | 5.22 |
| 1000 | 17.5 | 9.39 | 12.083 | 12.49 | 8.27 |
| 10000 | 8.1 | 11.70 | 11.48 | 11.98 | 11.34 |
| 100000 | 9.8 | – | – | – | 13.61 |
| Cost | Hybrid | Balas et. al | Kempka Kennington Zaki | Zaki | Kennington Wang |
| 100 | 33.5 | 5.52 | 3.813 | 1.55 | – |
| 1000 | 50.3 | 23.20 | 255.56 | 257.2 | – |
| 10000 | 45.3 | 30.09 | 32.96 | 32.8 | – |
| 100000 | 36.3 | – | – | – | – |

Table 10: Comparison of the hybrid auction code on a 32K machine with other parallel codes. Times are given in seconds; the top table is $n = 1000$ and the bottom is $n = 2000$. We compare with the Jacobi auction code results of Kempka, Kennington and Zaki, the Gauss-Seidel auction code results of Zaki, and two SAP codes. These codes are discussed in the introduction. An entry with a $--$ indicates those researchers do not report results for that range.

and most of the machine sits idle while the last paths are completed. Further, the methods we used to do each Gauss-Seidel bid at vp-ratio 1 do not transfer to the shortest paths problem. However, if a fast shortest paths code could be developed, a hybrid algorithm that uses the auction algorithm to assign all but a small number of people and then completes the assignments via shortest augmenting paths might be competitive or superior to our hybrid algorithm. The shortest augmenting paths codes that we discussed in the introduction are a similar hybrid approach.

Massive Parallelism is best exploited in problems where there is little complex communication and a huge amount of data that must be processed at every moment. Our experiences with the auction algorithm as a method to solve the assignment problem indicate that it does not fall into this paradigm. This is not an isolated phenomenon, but appears in a number of combinatorial approaches to optimization problems [32]. We have, however, established several methods that can bring combinatorial techniques closer to fast implementations on massively parallel architectures. It would seem that currently the best approach is to utilize massive parallelism while the problem continues to be massively parallel, and then to switch to another technique that is better suited to the tail of the problem. We have yet, however, to produce CM techniques for this problem that are significantly better than smaller scale MIMD machines. It is a challenging open problem to more fully exploit massive parallelism in this field.

**Acknowledgments**

Cindy Phillips wrote the first Jacobi version of the algorithm on the Connection Machine and we are grateful to her for helpful advice. We would like to thank Wati Taylor and Alan Edelman for useful suggestions about the implementations, and Jill Mesirov and David Shmoys for their encouragement. We would also like to thank Dimitri Bertsekas for several useful comments on a draft of this paper. We are grateful to Ralf Santos and the entire MCSG group at Thinking Machines Corporation for invaluable technical assistance. A preliminary version of part of this paper appeared in the proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation, October 1990 [33].

## References


[1] R. K. Ahuja and J. B. Orlin. New scaling algorithms for the assignment and minimum cycle mean problems. Sloan Working Paper 2019-88, MIT, Cambridge, MA, 1988.

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

[3] D. P. Bertsekas. A distributed algorithm for the assignment problem. Technical report, Laboratory for Information and Decision Sciences, MIT, 1979. Working Paper.

[4] D. P. Bertsekas. The auction algorithm: A distributed algorithm for the assignment and other network flow problems. *Annals of Operations Research*, 14:105–123, 1988.

[5] D. P. Bertsekas and D. Castanon. Parallel synchronous and asynchronous implementationsof the auction algorithm. Technical Report TP-308, Alphatech Inc., Burlington MA, November 1989.

[6] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, 1989.

[7] D.P. Bertsekas and J. Eckstein. Dual coordinate step methods for linear network flow problems. *Mathematical Programming*, 42:202–243, 1988.

[8] Guy Blelloch. Scans as primitive parallel operations. In *Proceedings International Conference on Parallel Processing*, pages 355–362, August 1987.

[9] L.D. Bodin, B.L. Golden, A.A. Assad, and M.O. Ball. Routing and scheduling of vehicles and crews. *Comp. and Oper. Res.*, 10:65–211, 1983.

[10] D. Castanon, B. Smith, and A. Wilson. Performance of parallel assignment algorithms on different multiprocessor architectures. Argonne National Lab. Report, in preparation, 1989.

[11] Y. Censor and S.A. Zenios. Massively parallel row-action algorithms for some nonlinear transportation problems. Working Paper 89-09-10, Decision Sciences Department, The Wharton School, University of Pennsylvania, 1989.

[12] Thinking Machines Corporation. Model CM-2 technical summary. Technical Report Report HA87-4, Thinking Machines Corporation, Cambridge, Massachusetts, April 1987.

[13] R.S. Dembo, J.M. Mulvey, and S.A. Zenios. Large scale nonlinear network models and their applications. *Operations Research*, 37:353–372, 1989.

[14] J. Eckstein. *Splitting Methods for Monotone Operators with Applications to Parallel Optimization.* PhD thesis, Dept. of Civil Engineering, MIT, Cambridge, MA, 1989.

[15] J. Eckstein. Implementing and running the alternating step method on the connection machine cm-2. Technical Report Working Paper 91-005, Harvard Business School, 1990.

[16] J. Eckstein and D.P. Bertsekas. On the Douglas-Rachford splitting method and the proximal point algorithm for maximal monotone operators. Technical Report CICS-P-167, Brown/Harvard/MIT Center for Intelligent Control Systems, 1989. To appear in Mathematical Programming.

[17] J. Eckstein and D.P. Bertsekas. An alternating direction method for linear programming. Technical Report Working Paper 90-063, Harvard Business School, 1990.

[18] A.M. Frieze, G. Galbiati, and F. Maffoli. On the worst case performance of some algorithms for the asymmetric travelling salesman problem. *Networks*, 12:23–39, 1982.

[19] A. V. Goldberg. *Efficient graph algorithms for sequential and parallel computers.* PhD thesis, MIT, Cambridge, MA, January 1987.

[20] A. V. Goldberg, S. A. Plotkin, and P. M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 174–185, 1988.

[21] A.V. Golderg, S. Plotkin, D.B. Shmoys, and E. Tardos. Interior point methods in parallel computation. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science.* IEEE Computer Society, October 1989.

[22] M.R. Hestenes. Multiplier and gradient methods. *JOTA*, 4:303–320, 1969.

[23] D. Hillis. *The Connection Machine.* MIT Press, 1985.

[24] R. Jonker and T. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38:325–340, 1987.

[25] R.M. Karp. Probabilistic analysis of partitioning algorithms for the travelling salesman problem in the plane. *Mathematics of Operations Research*, 2:209–224, 1977.

[26] 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. Technical Report OR-89-008, Department of Mechanical and Industrial Engineering, University of Illinois, Champaign-Urbana, 1989.

[27] J. Kennington and Z. Wang. Solving dense assignment problems on a shared memory multiprocessor. Technical Report 88-OR-16, Dept. of Operations Research and Applied Science, Southern Methodist University, October 1988.

[28] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[29] K. Mulmuley, U.V. Vazirani, and V.V. Vazirani. Matching is as easy as matrix inversion. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 345–354, 1987.

[30] Cynthia Phillips and Stavros A. Zenios. Experiences with large scale network optimization on the Connection Machine. In *Impact of Recent Computer Advances on Operations Research.* Elsevier Science Publishing Co., New York,NY, 1989.

[31] M.J.D. Powell. A method for nonlinear constraints in minimization problems. In *Optimization.* Academic Press, 1969.

[32] Joel Wein. Experience with minimum-cost circulation on the connection machine. Internal Thinking Machines Corporation Report, 1989.

[33] Joel Wein and Stavros Zenios. Massively parallel auction algorithms for the assignment problem. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 90–99, 1990.

[34] H. Zaki. A comparison of two algorithms for the assignment problem. Report ORL 90–002, Dept. of Mechanical and Industrial Engineering, University of Illinois at Urbana-Champaign, 1990.

[35] S.A. Zenios. Parallel numerical optimization; current status and an annotated bibliography. *ORSA Journal of Computing*, 1:20–43, 1989.

[36] S.A. Zenios. Matrix balancing on a massively parallel connection machine. *ORSA Journal On Computing*, 2:112–125, 1990.

[37] S.A. Zenios. On the fine-grain decomposition of multicommodity transportation problems. Working paper, Decision Sciences Department, The Wharton School, University of Pennsylvania, 1990.

[38] S.A. Zenios and R.A. Lasken. Nonlinear network optimization on a massively parallel connection machine. *Annals of Operations Research*, 14:147–165, 1988.

[39] S.A. Zenios and S. Nielsen. Massively parallel algorithms for singly constrained nonlinear programs. Working Paper 90-03-01, Decision Sciences Department, The Wharton School, University of Pennsylvania, 1990.

[40] S.A. Zenios and S. Nielsen. Massively parallel row-action algorithms for nonlinear stochastic network programs. Working paper, Decision Sciences Department, The Wharton School, University of Pennsylvania, 1990.