The Assignment Problem on Parallel Architectures

Martin L. Brady
Dept. of Electrical and Computer Engineering,
Pennsylvania State University,
University Park, PA 16802

Ken K. Jung, Mark V. LaPolla,
H.T. Nguyen, and Raghu Raghavan[1]
Advanced Computing Laboratory
Lockheed Research and Development Division
Palo Alto, CA 94304

Ramesh Subramonian
Division of Computer Science,
University of California,
Berkeley, CA 94720

Abstract

Implementation results are presented for new algorithms that solve the Assignment or minimum-weight bipartite matching problem in parallel on regular SIMD (Single-Instruction Multiple-Data) architectures. In particular, we are concerned with the implementation on simple bit-serial massively parallel arrays of processing elements (PE). For the Assignment problem, we have found several parallel algorithms that are good from both a theoretical and a practical point of view. We evaluate parallel implementations of three different algorithms. The first implementation is that of an early algorithm which was perhaps the first SIMD implementation described for Assignment. The second is more recent, and takes advantage of recent theoretical work on the problem for the PRAM model. Finally, as a result of discussions at the DIMACS workshop, we have implemented the Auction algorithm on a shared-memory multiprocessor, on SIMD machines, and on a distributed network of workstations.

[1]Current Address: Institute of System Science, National University of Singapore, Kent Ridge, Singapore 0511

1 Introduction

The Assignment problem is a classic combinatorial optimization problem. Its practical importance has resulted in a lot of theoretical algorithms and practical implementations.

We have found several parallel algorithms that run efficiently on SIMD machines and are good from both a theoretical and a practical point of view. SIMD machines consisting of many simple processors interconnected in a scaleable and regular manner have become commercially widespread. How restrictive this model of computation is, in practice, has been a focus of some of our research. We have designed, implemented and evaluated two different SIMD algorithms. One is based on Munkres's [Mun57] algorithm and was possibly the first SIMD implementation for the assignment problem. The other [Bra88] takes advantage of recent theoretical work [GPV88] on this problem for the PRAM model. Furthermore, as a result of discussions at the DIMACS workshop, we chose to implement the Auction algorithm [Ber91] on a variety of parallel architectures. We detail our insights into the behaviour of the algorithm and the idosyncracises of the various architectures we used.

The paper is organized as follows. In Section 2, we state the Assignment problem, describe the SIMD model and give details of the hardware platforms on which we obtained performance results. In Section 3, we describe the algorithm based on a parallelization of Munkres's algorithm. In Section 4, we describe an SIMD algorithm based on the more recent theoretical work for the PRAM model. Section 5 describes our experiences with the Auction algorithm on a variety of parallel architectures.

2 Definitions and Models

2.1 The Assignment Problem

The Assignment problem we consider is as follows: Let n be a positive integer, and let $|n|$ be $\{0, 1, \ldots, n-1\}$. Let $w : |n|^2 \Rightarrow Z$ be a matrix with finite precision entries. Find a permutation $\pi : |n| \Rightarrow |n|$ such that $\sum_i (w(i, \pi(i)))$ is a minimum.

This problem is identical to the (weighted) bipartite graph matching problem defined as follows. $G = (V, E)$ is an undirected bipartite graph such that V can be paritioned into two disjoint sets S and T and all edges have one endpoint in S and the other in T. Let the number of edges $|E| = m$ and the number of vertices $|S| = |T| = n$. A matching, M, is a vertex-disjoint subset of edges of E. The maximum cardinality bipartite matching problem is to find a matching in G of maximum size. In weighted bipartite matching, a weight, $w(i, j)$ is associated with each edge (i, j), and the cost of a matching is the sum of the weights of its edges. The goal is to find a maximum matching of minimum cost, and this is the same as the Assignment problem. The generalizations to (i) rectangular matrices, (ii) real numbers, and (iii) obtaining the maximum instead of the minimum are straight-forward and will not be considered further, since these are not directly relevant to the DIMACS challenge.

2.2 The SIMD Computing Model

The SIMD model consists of an array of identical processors (PEs) operating in lock-step synchrony. In each cycle, they perform the same instruction on their own data. In one step, a processor can perform one instruction or send/receive one bit/nibble from a neighbouring processor. Typically, the PEs have a limited instruction set and a very small finite precision (typically 1 to 4 bits). Thus, the execution time of an operation is dependent upon the word length. An $n \times n$ mesh-connected array contains n^2 processors positioned on an n-row, n-column square lattice. Each processor can communicate with its four nearest neighbours, except those on the border, which have only two or three neighbours. (Our algorithms could also be implemented on a mesh with "wraparound" connections, yielding a speedup of a small constant factor.)

The first commercially available chip embodying this architecture was the GAPP invented by W. Holsztynski [Hol86]. Our early results on the Assignment problem were implemented on an internally constructed machine AP2S (Adaptive Parallel Processing System, see [RJN90]]) based on the GAPP chip. The AP2S contains a two-dimensional (108 x 96) array of approximately 10,000 processing elements running at 10 MHz. Each processing element contains only 128 bits of local memory. Our algorithms were constructed for an "enhanced" SIMD model, in which the existence of row and column highways is assumed. Specifically, a row-OR or column-OR is assumed to take only one cycle. (This type of hardware enhancement is present in the AMT DAP machine for all the sizes manufactured, and we expect that our implementations for this model would also be efficient on the DAP [Red78]. It is, however, clearly an assumption that cannot hold in the limit for very large n.)

The AP2S system has limitations which restricted its usefulness for further investigations for the DIMACS project. Primarily, the limited local memory severely restricted the size of problem which was feasible. Thus, the remainder of our implementations (i.e. for larger arrays) were done on the commercial Wavetracer DTC machine (see [Jac90]) and MasPar Machines (see [Bla90], [Nic90]). In the Wavetracer DTC's largest configuration, there are 16K simple bit-serial processors with 2Kbits of on-chip memory and 32 Kbytes of off-chip memory per PE, each running at 11 MHz. While the hardware interconnection lattice is three-dimensional, our implementations configured the machine to a two-dimensional array (and to a one-dimensional array for the Auction algorithm). The DTC does not contains row and column highways. However, it does have a global bus that can perform the NAND of a bit from each PE in one cycle. This can be used to compute a global maximum.

The 2-D PE array of MasPar consists of 4-bit ALU running at 14 MHz and each PE can communicate with its 8 nearest neighbours through XNET. A 3 stage crossbar switch is provided for global routing, and some floating-point assisted hardware is also included at each PE. There are 192 bytes of on-chip register and 16 Kbytes of fast off-chip memory per PE. A global-OR (GOR) hardware is provided as well. The above features impact the performance on the Assignment problem in the following ways. The 4-bit ALU takes a quater of the cycle count as the bit-serial ALU array if the wordlength of operands is a multiple of 4. The global-OR hardware only gives an edge in global maximum/minimum, but not on column/row maximum/minimum. The software language used (MPL, MasPar Parallel application Language [Chr90]) required the programmer to develop the partition and virtualization if the data size was larger than the array size.

It is important to note that some of our results have a computer hardware/software flavor rather than a purely mathematical one. For example an early invention of ours called the Distributed Macro Controller (DMC) [HR88] allows, among other things, a run-time allocation of memory. This significantly enhances the speed of execution of the algorithm as discussed in the next section. We are also interested in the bit-level performance of these implementations, since massively parallel machines generally use bit/nibble serial processors. Further, the bit complexity is an inherently important quantity in the cost scaling techniques such as in [GPV88]. The architectures we have chosen represent the simplest and least expensive massively parallel computing models (around $10 per PE, total systems cost of a hundred thousand dollars for about 8,000 processors in the Wavetracer). They represent a different way to arrange about the same amount of hardware as a high-end workstation, as opposed to a multimillion dollar supercomputer. Thus, we view our results as an evaluation of the amenability of Assignment problem to this alternate architecture, and believe a comparison to sequential implementations on high-end workstations would be valid and interesting.

3 Munkres' Algorithm

3.1 The Munkres Modification of the "Hungarian" Solution

J. Munkres cited the following outline of Kuhn's algorithm for the Assignment problem:

1. Subtract the smallest element in w from each element of w, obtaining a matrix w' with non-negative elements, and at least one zero.
2. Find a minimal set S' of "lines" (rows and columns) of w', n' in number, which contain all the zeroes of w'. If $n' = n$, there is a set of n independent zeroes of w' and the elements of w' in these n positions are the desired solution. (The positions of the zeroes are called independent if no two of them belong to the same row or column of the matrix).
3. If $n' < n$, let h' denote the smallest element of w' which is not in any line of S'. Then $h' > 0$. For each line in S', add h' to each element of that line; then subtract h' from each element of w'. Call the new matrix w''.
4. Repeat Steps 2 and 3, using w'' in place of w'.

Munkres improved upon Step 1, which though performed only once, has a prolonged impact upon subsequent steps. However, his main contribution was in solving for Step 2. Our parallelization of the Munkres steps involves what we might call both "local" as well as g"lobal" parallelizations [Hol87, Ran87]. By "local" we here mean faithfully following the sequential algorithm, and parallelizing the steps as far as possible. However, we also make some changes that allow a single parallel step to correspond to several simultaneous Munkres steps. Each one of the latter, in turn, allows a "local" parallelization. We do not intend to attach any geometric meaning to the terms "local" and "global" here; these refer only to the structure of the algorithm. The algorithm actually coded on the AP2S is described in detail in [Hol87]. We now describe the performance on an "enhanced" AP2S simulation, which would be very similar in performance to that on a DAP [Red78].

3.2 Performance of the Parallel Munkres-style Algorithm

Before displaying the results of our algorithm, we make a few remarks. The sequential Munkres algorithm may require $O(n^3)$ time. The performance of our parallelization can be shown in the worst case to be $O(n^2 \log nC)$ with n^2 bit-serial processors and weights with a magnitude $< C$, (so that $\log C$ is the number of bits needed to specify the weights, and $\log n$ bits are required to represent a processor's row or column number). Thus, the parallel work (processor-time product) is $n^4 \log nC$, which appears poor. However, several points should be mentioned:

1. With random inputs, the running time is dictated by average-case results. We have not analyzed this rigorously. If the approximation that the weights remain random through the Munkres modifications is reasonable, then the average-case time bound may be shown to be $O(n \log nC)$.

2. Since the weights are decreasing with the repetition of the main steps of the algorithm, a machine that offers dynamic memory allocation, or deals with only as many bits as necessary at each step without overhead, offers an advantage. Such is the case with the machine augmented with the DMC that we constructed. In this case the average running time is less than that indicated by 1. above.

3. The constants hidden within the asymptotic bounds are quite small, especially for the enhanced GAPP model we are considering. This model allows the row-OR column-OR to occur in one cycle.

We now justify the above assertions by displaying the results of the runs in Table 1. In all cases, we display the running time in terms of machine cycles. The AP2S runs at 10 MHz, thus 10,000 cycles = 1 msec. The cycle count does not include any input-output time; further the cycle count assumes the "enhanced" GAPP model, in that a row-OR or column-OR is counted to take only one cycle. The x-axis counts the number of nodes as specified by the DIMACS instructions. Thus 64 nodes means that the matrix for the Assignment problem is 32×32, or in the language of graph matching there are 64 nodes divided into two sets of 32 each. Table 1 shows the averaged cycle-count (constructed as specified by the DIMACS consortium) for (i) random 11-bit samples (obtained using "assign.c" in the DIMACS library), (ii) 21-bit numbers (obtained using "dcube.c") and (iii) $i \times j \times 100$.

We feel that the times are surprisingly fast. Even for the 256×256 matrix with all entries having 21-bits, the algorithm completes (with the caveats mentioned above) in less than 40 msec on the average, with a 10 MHz clock. (It should be pointed out that more modern chips or machines run at higher clock speeds). For the 21-bit case, the times have dropped to about 16 msec. We should note that our actual array size is approximately 100 x 100, so that most of the time would be spent in input and output for the larger sizes of problem, due to the very limited amount of local memory per processor. As stated, this has not been included in the cycle count. The small size of our array is also why we are not reporting larger sized problems.

Size	*Cost*	*Time (in msec)*	*Instructions (in 1000s)*
16	1000	0.41	4.14
32	1000	1.03	10.34
64	1000	2.65	26.45
128	1000	6.56	65.64
256	1000	14.5	145.35
512	1000	34.1	341.03
1024	1000	75.1	750.58
16	1000000	0.68	6.78
32	1000000	1.90	19.05
64	1000000	5.27	52.69
128	1000000	14.0	139.6
256	1000000	36.5	365.1
512	1000000	94.6	946.2
1024	1000000	239	2386
16	$i \times j \times 100$	3.20	31.97
32	$i \times j \times 100$	18.4	184.1
64	$i \times j \times 100$	110	1104

Table 1: Performance of Munkres' algorithm on AP2S

4 Parallel Cost-scaling Algorithm

4.1 Description of the Algorithm

Recently, we have designed Assignment algorithms for SIMD implementation, building upon theoretical advances in algorithms based on the PRAM model [Bra88]. The technique is an adaptation of the sublinear time parallel PRAM algorithm of [GPV88], which uses a two-phase cost scaling approach. We obtain an $O(n^{5/3} \log^2 nC)$ time algorithm, where weights are integral and C represents the weight of maximum magnitude (one extra factor of $\log nC$ is incurred by the cost scaling iterations). The best previous SIMD algorithms appear to have been $O(n^2 \log nC)$ on a bit-serial $n \times n$ mesh. This algorithm is fairly memory efficient, using only a small constant number of $\log n$-bit registers per processor. To adapt these ideas to the strict SIMD model, we employ a novel technique in which the data is permuted in each pass to make optimal use of the local communication.

A step-by-step description of the algorithm is given below. The outer loop of the algorithm performs cost scaling, in which a solution is obtained through iterative improvement. Steps 1 through 4 in "Match()" (Figure 1) begin with a solution which is 2 ϵ-optimal and improve it to an ϵ-optimal solution (see [Ber86] for a discussion of the cost scaling technique). The algorithm has two major phases. The first is a very efficient "Push and Relabel" strategy which takes one step on many augmenting paths in parallel. The second is a more powerful but slower (in constant factor) routine which searches for complete disjoint augmenting paths. Each pass of the second phase finds all of the one-step paths of the first phase, some multiple-step paths, and at least one complete augmenting path. Thus, while each pass of the second phase takes longer, it may require fewer total passes. Theoretically, the first phase is more efficient at the beginning when there are many free vertices, and later the second phase becomes more efficient because it is guaranteed to increase the matching size by at least one during every pass. There is, therefore, an optimal "cutoff" point at which one should switch from phase 1 to 2.

Let us describe the computation required in each phase. Phase 1 is described below as "Push_Relabel_Matching" (Figure 2). Phase 1 begins by marking eligible edges from free S-vertices, using simple row and column operations. Next, a maximal matching of eligible edges is computed, by sweeping the matrix diagonally from the top left to the bottom right. (This subroutine is not much more time consuming than the row/column operations.) Finally, after selecting new matched edges and unmatching the old matched edges with which they conflict, we update the dual variables. This consists of increasing the costs in each column until the column's matched edge has cost $= \epsilon$. We then decrease the costs in each row until the cost of some unmatched edge in the row is $= -\epsilon$. Again, these require only simple row and column operations. Each step is $O(n)$, times the bit length of the operation, with very small constants.

The second phase, called "Shortest_Path_Matching" (Figure 3), works much harder at the dual variable update stage, finding all feasible augmenting paths. In place of the maximal matching step, this allows us to take as many steps as possible along augmenting paths from the free S-vertices. Furthermore, by reordering the rows and columns, we can find these augmenting paths in one diagonal sweep of the matrix, in time not much greater than a simple maximal matching. The time-consuming portion of phase 2 is in finding all feasible augmenting paths. The heart of

```
Match( )
{
    ∀i∀j : c(i, j) ← n × w(i, j)
    ε ← C
    while (ε > 1)do
        Step 1:     Find the minimum unmatched edge cost C_j in each column j.
        Step 2:     Subtract C_j from all c(i, j) in each column j.
        Step 3:     Unmatch arcs (i, j) with c(i, j) ≥ ε.
        Step 4:     while there are unmatched vertices (number of matches < n) do
                        if ( number of matches < Crossover )
                            CALL Push_Relabel_Matching( );
                        else
                            CALL Shortest_Path_Matching( );
                        endif
                    endwhile
        ε ← ε/2;
    endwhile
}
```

Figure 1: Procedure Match()

```
Push_Relabel_Matching( )
{
    Step 4.1:   matched_cost(i, j) ← column broadcast of the cost of the matched edge.
                (If the column is unmatched, broadcast ε.)
    Step 4.2:   For all unmatched edges, mark eligible if c(i, j) < matched_cost(i, j).
    Step 4.3:   Find a maximal matching of the eligible edges.
    Step 4.4:   Column broadcast the newly matched edges new_M.
    Step 4.5:   Unmatch the old edges from newly matched columns.
                Match the new edges in these columns.
    Step 4.6:   Modify the column costs (increase them as high as possible without making the
                matched edge in the column have c(i, j) > ε). Column broadcast the cost of the
                newly matched edges.
    Step 4.7:   Modify the row costs (decrease them as low as possible without making the
                unmatched edge in the row have c(i, j) < ε).
}
```

Figure 2: Procedure Push_Relabel_Matching( )

```
Shortest_Path_Matching( )
{
    Step 4.1:   Compute the distance labels (shortest paths to free T-vertices).
    Step 4.2:   Permute rows and columns to order S-vertices and T-vertices by
                distance label, with the largest distance labels on top/left.
    Step 4.3:   Mark eligible edges (those which appear in shortest augmenting paths).
    Step 4.4:   Set reduced costs cb(i,j) as:
                cb(i,j) ← c(i,j) − d_s(i) + d_t(j) where arcs (i,j) are unmatched.
                cb(i,j) ← c(i,j) − d_t(j) + d_s(i) where arcs (j,i) are matched.
    Step 4.5:   Find and augment along maximal S-paths, starting at the largest distance labels.
    Step 4.6:   Set c(i,j) ← cb(i,j).
}
```

Figure 3: Procedure Shortest_Path_Matching( )

Size	*Time (in sec)*
8×8	0.02
16×16	0.07
32×32	0.31
64×64	1.11
128×128	6.37

Table 2: Push Relabel Matching Algorithm ($w(i,j) = i \times j$)

the computation is the calculation of the shortest path from every S-vertex to some free T-vertex. Theoretically, by carefully pipelining the operations, such a distance computation can be done in $O(n \log nC)$ steps on a bit-serial $n \times n$ array [Sco80], and our implementation achieves this bound. Therefore, phase 2 has the same asymptotic complexity as phase 1. However, the constant for the distance computation is nearly two orders of magnitude greater than that of the row/column operations, which limits its competitiveness.

4.2 Performance Results of the Parallel Cost-Scaling Algorithm

We have implemented the above algorithm on the Wavetracer, and report some results (Tables 2, 3, 4 and 5) for each of the two major subroutines (in msec) and for the entire algorithm (in sec) below. These results are for random data in the range $0 - 1000$.

Note that the second phase (distance) is about 50 to 60 times slower than the first. For the random data tested, the average time was faster when using only phase 1 to completion, rather

Size	*Time (in sec)*
8×8	0.06
16×16	0.28
32×32	1.06
64×64	4.34
128×128	27.8

Table 3: Shortest Path Matching Algorithm ($w(i,j) = i \times j$)

128×128	*Time (in sec)*
Crossover at 115	7.3
Crossover at 121	6.1
Crossover at 125	3.9
Crossover at 126	3.7
Crossover at 127	3.4
No Crossover	6.4

Table 4: Combination of Shortest Path and Push Relabel Matching Algorithm ($w(i,j) = i \times j$)

Size	*Push_Relabel cycles*	*Distance cycles*	*Algorithm Cycles*
-	(in msec)	(in msec)	(in sec)
256×256	1.9	93	1.12
512×512	3.5	19	3.33
1024×1024	6.9	404	13.1
2048×2048	13.8	843	34.9

Table 5: Average Performance of the Push Relabel and Shortest Path passes, and of the entire algorithm for large input size (random 11-bit data)

than invoking phase 2. It appears that for some examples it may be advantageous to use the second phase, only for the very last matched edge (i.e., for one pass, to avoid a potentially very large number of passes of phase 1).

One particularly difficult problem is obtained by setting each cost $c(i, j)$ to $i \times j$. For this problem, we find that a 256-node instance requires 333 sec using "Push_Relabel_Matching" alone, and only 156 sec when crossing over to full distance calculation when all but one pair of nodes are matched. Crossing over sooner degrades this improvement, however.

The speed of these "theoretically good" algorithms does not appear competitive with the Munkres implementation of the previous section. However, it must be noted that we no longer have the use of row and column busses, providing row and column OR in one cycle. The Push Relabel pass would be much improved by row/column broadcast, since row/column minimum and maximum are heavily used. For the 16K processor, machine, this would mean a factor of 128 improvement in row and column operations. The shortest path matching algorithm could probably also be improved by using the busses. The majority of the time is spent in the distance computation, which, as written, uses only local communication. However, with broadcast we could write a much simpler non-pipelined algorithm with perhaps an order of magnitude improvement in the constant. A second source slowdown is in the cost scaling. To simplify implementation, we used only a scale factor of 2. Therefore, a great many cost scaling passes were required, often only to bring a few new edges into the lower epsilon range. For instance, consider a 1K by 1K problem with data in the range 0 to 1000. The number of passes with scale factor 2 would be about 20, while increasing the scale to just 10 would result in 6 passes. Finally, one critical difficulty with these algorithms is that they require more storage per processor than the Munkres algorithm. This limits the size of problem which can be efficienty run on the parallel array, and is a reason why we have added the Auction algorithm to our implementations, as discussed in the next section.

5 The Auction algorithm

5.1 Description of the Auction algorithm

We refer the reader to [Ber91] for an excellent tutorial. We provide a brief description of the algorithm. We start with a weight matrix w where $w(i, j)$ is the cost of assigning person i to object j. Initially, all persons are unassigned. Each object, j, has a price, $price(j)$, initially 0. The Auction algorithm proceeds in "rounds" or "bids". In a bid, an unassigned person bids for the object, j_i with maximal value m_i where $m_i = max_j\{w(i, j) - price(j)\}$. It also computes the second best object value, $nm_i = max_{j, j \neq j_i}\{w(i, j) - price(j)\}$. It then raises the price of j_i by $m_i - nm_i + \epsilon$, unassigns any person assigned to that object and assigns itself to that object. This process is repeated until all objects have received at least one bid. If $\epsilon < \frac{1}{n}$ and $w(i, j)$ are integers, then the assignment obtained upon termination is optimal. We use ϵ-scaling in our implementations, which reduces the number of bids. ϵ-scaling consists of repeating the algorithm several times, starting with a large value of ϵ and progressively decreasing it until it is less than $\frac{1}{n}$ for integer costs. Also, for a given ϵ, rather than wait for all objects to be matched before we reduce ϵ, we set a threshold. Once the number of unmatched objects falls below this threshold,

Problem size	*Fraction of bids*	*Number unassigned*
2048	34.1%	≤ 32
2048	25.4%	≤ 16
2048	16.3%	≤ 8
2048	11.7%	≤ 4
2048	7.7%	≤ 2
2048	4.9%	≤ 1
4096	31.6%	≤ 32
4096	23.4%	≤ 16
4096	16.5%	≤ 8
4096	12.4%	≤ 4
4096	9.6%	≤ 2
4096	6.7%	≤ 1

Table 6: Fraction of bids with a given number of unassigned persons

we move to the next ϵ value. The threshold is gradually reduced until it becomes 1 when ϵ is 1.

Combined Forward-Reverse Auction. [BCT91] proposed a variation on the Auction algorithm called Reverse Auction, where, roughly, *objects* compete for persons by lowering their price to a level that is sufficiently low to lure a person away from his currently held object. They showed that a judicious combination of Forward and Reverse Auction solved problems faster than either one alone could.

We decided against this variation despite its claimed superior performance [Cas91] for a simple reason: memory accesses. The weight matrix is stored as columns (FORTRAN) or as rows (C or Pascal). The Forward-Reverse Auction accesses the matrix row-wise as well as column-wise. Given that most memory systems favour block transfers, it seemed unwise to access columns of a matrix that had been stored row-wise, especially if the matrix is so large that it needs to be paged in from disk.

Gauss-Seidel vs. Jacobi. The variation of the Auction algorithm when only one person bids at a time, is called the *Gauss-Seidel* implementation. If all unassigned persons bid at the same time, we get the *Jacobi* implementation. Both yield correct results. The Jacobi implementation is more parallel but requires more bids for termination. The decision to use the Gauss-Seidel variation of the Auction algorithm rather than the Jacobi variation was based on the following. (i) A reasonably high proportion of the total number of bids required by the Auction algorithm is spent with very few unmatched objects as shown in Table 6 and as discussed in [WZ91, BC89], and (ii) the Jacobi variation requires more bids than the Gauss-Seidel [WZ91, BC89]. Further, we suspect that as the problem size grows, the fraction of bids with very few unmatched persons increases.

5.2 Characteristics of the Auction algorithm

1. Processor-to-memory bandwidth. The weight matrix grows as $O(n^2)$. Therefore, memory requirements and processor-to-memory bandwidth are of concern.

2. The heart of the problem. The essential computational problem is finding the maximum of n numbers.

3. Locality of reference. The Auction algorithm does not exhibit any regularity of access patterns across bids. In other words, while the number of unmatched persons decreases rapidly, the identity of the persons in the small set of unmatched persons fluctuates rapidly and without any regularity.

4. Regularity of access patterns within a bid. The data necessary to calculate a bid are in contiguous memory locations.

5. Computation to communication ratio. The amount of computation done per datum read is extremely small, consisting of a few integer arithmetic operations and comparisons.

We analyze these considerations in the context of a variety of architectures. We present our strategy, rationale for that strategy, performance results, lessons learnt, and what architectural features are important.

5.3 SIMD Implementation

The Auction algorithm was implemented on an 8K MasPar MP-1208 and a 16K Wavetracer DTC-04032. The processor-to-local-memory bandwidth[2] is 2.2 Mbytes/sec on the Wavetracer and 1.64 Mbytes/sec on the MasPar. This figure is arrived at as $\frac{p \times 4}{t_{read}}$, where t_{read} = the time to read a 4-Byte integer from memory, p = number of processors. This bandwidth is sustainable until the problem becomes too large to fit into the memory of the processors, which is 16K processors $\times$ 32 Kbytes = $\frac{1}{2}$ Gbytes for the Wavetracer and 8K processors $\times$ 16 Kbytes = 128 Mbytes for the MasPar. Note that these machines do not perform block transfers. This did not affect our results because blocks transfers can help only when n exceeded p and memory limitations kept us from such large problems. While the processors have registers, they have no automatic cache mechanism. This is not a drawback for the Auction algorithm which displays no locality of reference across bids. The SIMD machine was configured as a 1-D array. The jth processor was assigned the jth column of the weight matrix, $w[1..n][j]$, $price[j]$, $s[j]$ and $t[i]$ where $s[j]$ is the s-vertex to which j is matched and $t[i]$ is the t-vertex to which i is matched. The built-in function to compute the maximum was used. A small part of each bid - determining which object had to be assigned to which person, unassigning objects and updating the price - was performed on the front end.

A large part of the problem consists of performing a global maximum, for which a built-in function is provided.

[2]The processor-to-local-memory bandwidth is the product of the rate at which a processor can access its local memory and the number of processors.

-	-	*MasPar (time in sec)*		*WaveTracer (time in sec)*	
Size	*Cost*	*Unscaled*	*Scaled*	*Unscaled*	*Scaled*
128×128	100000	0.95	1.24	25.1	24.27
256×256	100000	2.85	2.50	74.7	55.29
512×512	100000	7.65	4.62	198	117.99
1024×1024	100000	19.6	9.94	565	277.11
2048×2048	100000	48.3	20.43	1342	603.64
3072×3072	100000	71.8	32.53	2043	980.81
3800×3800	100000	82.1	40.69	2373	1237.43
128×128	10000000	0.83	1.30	23.5	27.33
256×256	10000000	2.34	2.51	61.4	57.84
512×512	10000000	6.98	5.04	189	131.67
1024×1024	10000000	20.1	10.28	575	290.32
2048×2048	10000000	75.5	21.76	2050	646.96
3072×3072	10000000	752	34.29	13027	1040.07
3800×3800	10000000	531	47.70	9074	1463.69
128×128	$i \times j$	118.81	2.47	3711.59	53.3
256×256	$i \times j$	906.12	9.57	29898.51	275
512×512	$i \times j$	10151.93	30.9	241913.29	931
1024×1024	$i \times j$	–	131	-	4008
2048×2048	$i \times j$	–	313	-	9707
3072×3072	$i \times j$	–	689	-	21629
3800×3800	$i \times j$	–	1463	-	44831
128×128	$i \times j \times 100$	115.91	2.46	3771.35	59.3
256×256	$i \times j \times 100$	919.56	9.58	30450.25	286
512×512	$i \times j \times 100$	–	29.5	-	910
1024×1024	$i \times j \times 100$	–	129	-	4041
2048×2048	$i \times j \times 100$	–	279	-	8938
3072×3072	$i \times j \times 100$	–	588	-	18907
3800×3800	$i \times j \times 100$	–	797	-	25867

Table 7: Results on SIMD Machines

Results and Discussion Our results are shown in Table 7.

We learned the following lessons:

1. The frustration of bit-serial computers. The often touted advantage of bit-serial computers does not translate into any advantage in practice. In commercially available bit-serial machines, the cycle-count for operations is a very poor indicator of actual computation time. We had recognized and solved this problem several years ago in our prototype, the AP2S [RJN90] with a Distributed Macro Controller. However, such a controller remains unavail-

Operation	*Loop Time*	*Repeated Time*
Off-chip Memory to On-chip Memory	1.38	4.96
Off-chip Memory to Off-chip Memory	0.54	1.45
On-chip Memory to On-chip Memory	2.25	8.28
32-bit addition	6.33	12.45

Table 8: Time to perform specified operation on the Wavetracer (in μsec)

Operation	*Loop Time*	*Repeated Time*
Memory-to-register	6.42	20000
Memory-to-memory	12.12	40000
Register-to-register	3.61	10000
32-bit addition	20.0	240

Table 9: Time to perform specified operation on the MasPar (in μsec)

able in commercial machines. For example, consider a 4-byte addition, $a \leftarrow b + c$, performed n times in a loop or simply replicated n times. One would not expect a difference in the time taken by these two dummy programs. On the contrary, there is a significant difference. This slowness is due to the controller bottle-neck. To understand this, let us explain how, for example, the Wavetracer works.

On the Wavetracer, a program compiled by multiC runs on the host which sends appropriate commands to the 16 MHz command controller via a SCSI bus. The controller generates array processor microcode instructions and sends it to the sequencer. Then the sequencer executes these instructions on the array processors. Once the instructions are executed, the sequencer waits for the controller to generate the next set of microcodes. The controller waits for the host to generate the next command. The time taken for these operations is substantial. The array processor is poorly utilized unless it is executing a long operation (eg., floating-point multiplcation) such that the computation time subsumes the time for the decoding and broadcast process. Table 8 shows the disparity between the loop time and the sequential time. The disparity decreases as the operations become more complex. MasPar has a similar problem as Table 9 indicates.

Another slowdown arises as follows. The flow of information from the host to the array processors is pipelined. Whenever this pipe is broken, a delay is incurred. Consider the code segments in Figure 4 to find the most desired object i.e., the processor with the highest v value. Variables $maxi$ and j are resident on the host while v, w, $processor\_id$ and p are resident on the array processors. The code on the left is about 1.25 times slower than the code to the right. This is because in assigning the maximum to $maxi$, a variable resident on the host, we are forcing the host to wait for the value of $maxi$ and hence are breaking the instruction pipeline.

code for finding j: object with maximum $v = w[i] - p$

$v \leftarrow w[i] - p$
$maxi = >?= v$ { $=>?=$ is the max-finding operator }
if $maxi == v$ **then**
    $j = <?= processor\_id$
endif

$v \leftarrow w[i] - p$
if $v == (>?= v)$ **then**
$j = <?= processor\_id$
endif

Figure 4: Effect of breaking pipeline in SIMD machines

2. The ability to compute an associative function. The time take to perform a 4-Byte global maximum on the MasPar is 0.51 msec and 0.34 msec from memory and register (benchmark using MPL compiler) respectively and Wavetracer is 2.9 msec (multiC compiler), instead of 326 cycles or 32.6 μsec on the AP2S which utilized the Global-OR feature of the GAPP. Global-OR of a 4-Byte word on the MasPar takes 0.1 msec and 0.07 msec from memory and register respectively, and 2.9 msec on Wavetracer. The global maximum operation finds the maximum price and the next maximum price per person, and Global-OR is used to locate the found maximum price.

   The average time per bid for $n = 2048$ was 2.5 msec on the SPARCstation-2, 1.241 msec on the MasPar and 9 msec on the Wavetracer. The times for the MasPar and Wavetracer would remain constant until $n > p$, after which virtualization would introduce an overhead. For $n = 2048$, we achieved a speedup of 2 on the MasPar slow down of 6 on the Wavetracer, compared to the SPARCstation 2.

3. The imbalance between communication and computation. While the work of [WZ91] is noteworthy, we feel that the Auction algorithm is mismatched on an SIMD architecture. It is interesting to note [WZ91]'s conclusion that "we have yet, however, to produce CM techniques for this problem that are significantly better than smaller scale MIMD (Multiple-Instruction Multiple-Data) machines." For example, the MasPar can perform 400 million integer additions per second. However, it can read only about 0.4 million integers per second from memory.

   The Auction algorithm makes a strong case for an MIMD machine like the CM-5 or Intel Delta, *provided that they provide low latency communications*. One of the guiding principles in the design philosophy of the CM-5 was to leverage the technology advancements in the "killer-micros". The Auction algorithm uses relatively simple instructions and has the structure necessary which makes it amenable to techniques like pipelining, block transfers and hence, an ideal candidate for powerful RISC-based processors. The performance of Auction on a single processor degrades only when we have to go out to disk repeatedly. The processor-to-local-memory bandwidth shortfall could be redressed by having many processors, each with their own communication route to memory. Since the weight matrix, which is responsible for the bandwidth shortfall, is immutable, this suggests that there is a rationale for designing of read-only memory which permits simultaneous reads and does not serialize concurrent accesses as a typical memory interface does.

4. For memory intensive problems such as Auction, the time to load the data often substantial (Table 10).

Problem size	*Time to transfer (in sec)*
512×512	.072
1024×1024	17
2048×2048	80
3072×3072	148

Table 10: Time to transfer data from a binaty file on the DEC workstation- host into the memory of MasPar array processors.

5.4 Shared Memory Implementation

We implemented the algorithm on a 4-processor workstation- from Silicon Graphics, the IRIS 4D/340 DVX, chosen because it was representative of shared memory systems (and because it was the only machine on which we could hog vast amounts of CPU and memory without risking the ire of other users!). It has four 33 MHz R3000 RISC based processors from MIPS with 48 Mbytes of ECC memory and 2 Gbytes of disk. It has a primary cache of 64 Kbytes which is connected to the processor via a Processor Bus, which can transfer data at 8 bytes every clock cycle. An MPLink bus connects processors to memory. A second level data cache, 256 Kbytes in size, serves as a buffer between the first level cache and the MPlink bus. It provides block transfer capability for the MPlink Bus. The MPlink bus has 32 address lines and 64 data lines. An MPlink Bus transaction is six cycles in length, although the last two data transfer cycles can overlap with the first two cycles of the next transaction, resulting in a sustainable data transfer rate of 64 Mbytes/sec. This is substantially higher than the 2-4 Mbytes/sec on the SIMD machines. However, it drops to 2 Mbytes/sec when we have to go to disk through the SCSI.

In order to reduce the memory traffic, the columns of the weight matrix were partitioned among the processors. A single processor was designated as the master processor. It was responsible for deciding which person would bid and also for coalescing the partial results of the other processors. Since this required a simple read from shared memory, the latency for the flow of information was negligible. Had there been more processors, the work of coalescing bids would have been distributed among the processors.

We had essentially a producer-consumer relationship between the master processor and the rest of the processors. We chose not to incur the overhead of using system calls for the synchronization. Instead we used the following tagged scheme and busy-waiting (justified by the fact that we were running in single user mode and that the workload was evenly balanced, thereby making the waiting time minimal).

Assume that the master decides that the bid for person i must be calculated. Let this be the bth bid. It writes i into *which_person* and b into *tag_which_person*, both of which are shared locations. A slave processor knows which bid is to be calculated. It first reads *tag_which_person*. If this value is less than b, it abandons the data as premature. Note that it could not be greater than b. If it is equal to b, it starts calculating its portion of the bid for i. When it is done, it writes its partial results to a shared region and then writes the tag of b. The protocol of reading in the opposite order of writing to obtain consistent values was first suggested by Lamport [Lam77]. The ability to predict the tags of the values makes this a "knowledgeable" algorithm, as defined in [Sub92]. Lack of space prevents a formal proof that this protocol is correct.

Results and Discussion. The machine was used essentially as a distributed memory machine with no dynamic reallocation of work. The shared memory was used to provide a low-latency inter-processor communication mechanism. While a single slow processor could degrade the entire performance, this was not observed because of even load balancing. Contrary to what was expected, the MPlink bus was not a bottleneck. The bottleneck was the SCSI which connects main memory to disk, which came into play when the problem size was too large for the weight matrix to fit in main memory.

Figure 5 shows how the average time per bid fluctuates when $n = 4096$ and the weight matrix could not fit in main memory. Note how the change in ϵ causes an increase in the average time per bid. This is because when ϵ changes, all objects become unmatched. The average time per bid was 0.63 msec for $n = 2048$ and 0.31 msec for $n = 1024$. For $n \leq 2048$, the problem was resident in main memory and the time per bid was very uniform. Table 11 shows the results for the shared memory implementation.

Problem size -	Number of bids -	Average time per bid (msec)	Total time (seconds)	Range of random numbers
1024×1024	30464	0.31	9.44	100000
2048×2048	66360	0.63	41.82	100000
4096×4096	131151	55.64	7297.8	100000

Table 11: Performance results on shared memory system

A Note on Asynchronous Implementations We chose not to use asynchronous variations of the Auction algorithm despite the [BC89] claim that "better asynchronous implementations outperform substantially the corresponding synchronous implementations on a shared memory machine." Asynchrony does not buy one anything. On the contrary, synchrony can cost one *when maintaining synchrony is expensive.* We found that maintaining synchrony did not induce

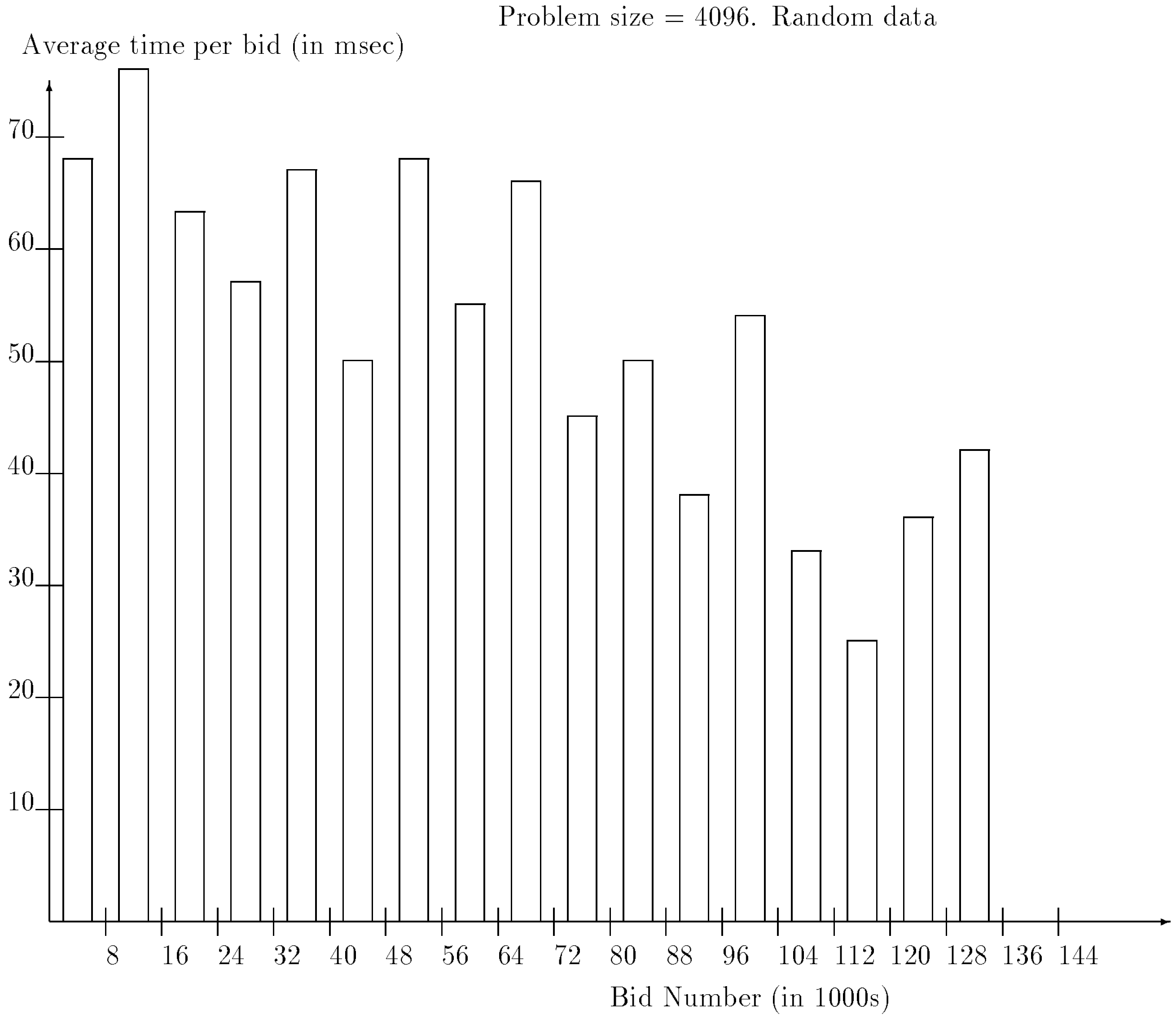

Figure 5: Variation of average bid time for large problem sizes

a performance penalty. It must be stated that the minimal overhead suffered to maintain synchrony is because of the relatively few numbers of processors used and the uniformity of the load distribution.

5.5 Distributed Implementation

Parallel Virtual Machine (PVM), a public-domain software package developed at Oak Ridge National Laboratories was used to implement a distributed solution [Sun91]. We believed the Auction algorithm to be amenable to a distributed solution for the following reasons:

1. One of the characteristics that makes a problem amenable to a distributed solution is that the problem descriptor should be small, the computing to be done should be large and the description of the solution should be small. This is in order that the communication costs

can be subsumed by the computing time.

One processor was designated as the master. The problem descriptor, the identity of the person bidding, was small. Broadcasting this information to 7 processors required roughly 1.5 msec. This is roughly the time taken by a SPARCstation-2 processor (running at 40 MHz and 28.5 MIPS) to perform one bid for $n = 1024$. Therefore, if $n = cp \times 1024$, then the overhead of the broadcast would be $\frac{1}{c}$.

Broadcasting is implemented in PVM using point-to-point messages with recursive doubling. Broadcast proceeds in "rounds", with the number of processors contributing to the broadcast effort doubling in each round. In round r, processor k transmits to processor $(k + 2^r) \bmod p$ and receives no acknowledgement. A processor k joins the broadcast effort at round r_k, where r_k = number of 1's in the binary representation of $k \bmod p$.

In the implementations of PVM, the UDP protocol was used [Pos80]. Therefore, the size (in bytes) of the data message, or datagram, exhanged is computed as follows: size = Ethernet header + IP header + UDP header + data + Ethernet trailer = 14 + 20 + 8 + data + 4 = 46 + data. Hence, to broadcast 8 bytes of data, 54 bytes of data are sent. With a 10 Mbits/sec Ethernet, this should take 41 μsec ideally.

2. The data could be partitioned easily by simply giving each processor some of the columns of the weight matrix. The weight matrix does not change during the course of the computation. The information to be updated in each bid, the price of the object for which a bid was submitted (if at all), is small. A further optimization was to combine this with the broadcast of the identity of the bidder. Since the cost of the broadcast is virtually unaffected by this additional information, this update introduced no additional overhead.

3. One of the often ignored but important factors that slow down distributed computations is load imbalance [Sun92]. There are two sources of load imbalance. One arises from the workload of other users on the different machines being used. This was not relevant to us, since we had the machines to ourselves. The second arises from computational imbalance. By partitioning the work in proportion to processor speeds, it was hoped that this imbalance could be redressed.

4. Since each processor had its own access to memory, we hoped to redress the processor-to-memory bandwidth.

Results and Discussion The Auction algorithm was implemented on 15 SPARCstation-2 processors connected on a 10 Mbits/sec Ethernet. We found that while there was no flaw in the rationale mentioned above, the performance was little short of disastrous.

Why was this so? The bottleneck was the "accumulate" or "gather" routine. In a bid, each processor, k, operates on the columns assigned to it. It calculates m^k, nm^k, $price[j]^k$, which are the values m, nm and $price[j]$ would have had had the columns assigned to k been all the columns of the weight matrix. Given these partial results, we needed to compute m, nm, $price[j]$ and k', the identity of the processor which has the object with the highest value (i.e., $price[j] = price[j]^{k'}$). Since the partial results of each processor could be coalesced using a binary tree (as is done in the broadcast), we felt that this should not take significantly longer than the broadcast time. Note that it would be overly optimistic to expect the same time as the broadcast since the broadcast consists of non-blocking sends whereas the accumulate involves both sends, receives and some small local computation. The rapid degradation in performance that the accumulate caused was unexpected.

Two approaches to performing the accumulate were tried: (i) Serial: All processors send their partial information to the processor designated as master, and (ii) Recursive doubling: The processors are laid out as the nodes of a binary tree and a node sends its partial information is sent to the parent. The performance results are presented in Table 12.

Number of Processors -	*Serial Accumulate*	*Recursive doubling Accumulate*	*Broadcast time* -
7	20.2	18.3	1.42
15	41.91	30.9	1.46

Table 12: Time (in msec) to accumulate partial results.

We came to two conclusions:

1. Given current hardware and software technology, the overhead associated with transmissions makes it impractical to run applications which require small (of the order of a few msecs) latency. This raises the following possibility to decrease latency. Any information broadcast on an Ethernet is received by all processors. Each message has the id of the receiving processor and a processor simply ignores a message that does not belong to it. This suggests designating an id for the Parallel Virtual Machine (PVM) and have a processor read all messages that had either its id or the PVM id.

2. Given that the memory is distributed, slow response from even one processor would degrade the entire system performance. There is simply no way to dynamically reassign work. This makes this approach increasingly treacherous as the number of processors is increased, unless one is in a position to guarantee the workload on *all* the participating processors.

   Note that the same criticism holds for an MIMD implementation (as on a CM-5 or Intel Delta). This does not hold true for a true shared memory system, where work can be dynamically re-allocated. Of course, reallocating work means re-distributing the data and it is not sure whether the necessary bandwidth would be available especially since the Auction algorithm is communication intensive i.e., the work done on each datum read is very small, thereby making it difficult to recover the cost of the communication.

6 Conclusions

Three algorithms have been designed on simple bit-level SIMD machines. A simple parallel algorithm for the Assignment problem, with theoretically sub-optimal speed, has been shown to be surprisingly fast in practice. Theoretically superior in the worst case, our cost scaling algorithm has difficulty keeping pace on average case data. This is because for randomly selected finite precision data, there are potentially many optima, and a simple algorithm (which may require many passes in the worst case) tends to converge to one of them rather quickly. Lacking, however, from the classes of input data are "real-world" data taken from practical problems, which are often decidedly non-random. It would be interesting to analyze the performances for such inputs, if it could be obtained.

We reached the following conclusions regarding the Auction algorithm. Its extreme simplicity makes it amenable to easy coding as well as keeps the constant factors, hidden in the big-O notation, small. The algorithm performs very little computation for each datum read. The computation power of most machines significantly exceeds their communication power (both processor-to-memory and processor-to-processor). A significant fraction of the bids have very few objects unmatched, a situation which seems to worsen as the problem size increases. Hence, we do not believe that it is amenable to "massively" parallel implementations as has hitherto been claimed. However, for reasonably small numbers of processors, it performs very well.

Our SIMD implementations squeezed the most out of the machine but came up short against the following inherent limitations: low memory-to-processor bandwidth, slow controller, breaks in the host-to-array-processor pipeline. Our shared memory implementations showed perfect speedup. However, they also showed that processor-to-memory bandwidth was the bottleneck when the problem could not be fit into the main memory. Our distributed implementations were an effort to redress the processor-to-memory bandwidth by trying to fit the problem into the main memory of many processors. While solving this problem, we ran into unacceptably high processor-to-processor latency.

The algorithm on the GAPP was written in assembler code, while the current Wavetracer implementation was written in MultiC in a few days. For reasonable comparisons we are concentrating on the machine cycles used by each of the parallel arrays. However, the controller at the front end of the array has also proven to be a major issue. In particular, the response times on the Wavetracer are much slower than the computation time. This is almost entirely due to the fact that the controller is a bottleneck, and is unable to decode instructions for the parallel machine to keep pace with the speed of the processor. This is in contrast to our controller, called the DMC and alluded to earlier, which could keep pace with the machine, and allowed subroutine calls, memory allocation, and reprogrammability all without any penalty in machine speed. Such controllers have been demonstrated by this investigation to be essential to realize the performance promised by massively parallel processors.

7 Acknowledgements

The work was performed with Lockheed Independent Research funds under project RDD506. We are also grateful to the people with Wavetracer, in particular Mary Cacciatone, Mike Lee, Tom McLaughlin and Robert Shuffleton. They made their machine available to us and gave freely of their time in advice and in assisting our debugging efforts. We would like to thank the people at MasPar, Don Rector in particular, for the use of their machine. Thanks to Mark Noga at Lockheed

PARL for his technical assistance. We also thank the referees of DIMACS for their comments, and for bringing the Auction method to our attention.

References

[BC89] D. P. Bertsekas and D. A. Castanon. Parallel Synchronous and Asynchronous Implementations of the Auction Algorithm. Technical report, M. I. T., 1989. To appear in Parallel Computing.

[BCT91] D. P. Bertsekas, D. A. Castanon, and H. Tsakanakis. Reverse Auction and the Solution of Inequality Constrained Assignment Problems. Technical report, M. I. T., 1991.

[Ber86] D. P. Bertsekas. A Distributed Algorithm for the Assignment Problem. In *Proceedings of the 25th IEEE Conference on Decision and Control*, 1986. MIT LIDS Tech Report P-1606.

[Ber91] D. P. Bertsekas. The Auction Algorithm for Assignment and Other Network Flow Problems: A Tutorial. *Interfaces*, 20(4):133–149, July-August 1991.

[Bla90] T. Blank. The Maspar MP-1 Architecture. In *Proceedings of the 35th IEEE International Computer Conference*, pages 20–24, March 1990. San Francisco.

[Bra88] M. Brady. Parallel Bipartite Matching on SIMD Arrays. Technical report, Lockheed Palo Alto Research Laboratory, 1988.

[Cas91] D. A. Castanon. Reverse Auction Algorithms for Equality and Inequality Constrained Assignment Problems. Technical report, Boston University, 1991. Accepted at the First DIMACS International Algorithm Implementation Challenge: Network Flows and Matching.

[Chr90] P. Christy. Software to support Massively Parallel Computing on the Maspar MP-1. In *Proceedings of the 35th IEEE International Computer Conference*, pages 29–33, March 1990. San Francisco.

[GPV88] A. Goldberg, S. Plotkin, and P. Vaidya. Sublinear-Time Parallel Algorithms for Matching and Related Problems. In *Proceedings of the 29th Symposium on Foundations of Computer Science*, pages 174–185, 1988.

[Hol86] W. Holszytnski. Geometric-Arithmetic Parallel Pocessor, GAPP, February 1986. Canadian Patent Number 1202208.

[Hol87] W. Holszytnski. The Assignment Problem on GSIMD Arrays. Technical report, Lockheed Palo Alto Research Laboratory, 1987.

[HR88] W. Holszytnski and R. Raghavan. The Distributed MAcro Controller for GSIMD Arrays. In Stuart K. Tewksbury, editor, *Concurrent Computations: Algorithms, Architecture and Technology*, chapter 35, pages 689–696. Plenum Press, 1988.

[Jac90] J. H. Jackson. The Data Transport Computer: A 3-D Massively Parallel SIMD Computer. In *Proceedings of IEEE COMPCON*, pages 264–269, Spring 1990.

[Lam77] L. Lamport. Concurrent reading and writing. *Communications of the Association for Computing Machinery*, 20(2):806–811, November 1977.

[Mun57] J. Munkres. Algorithms for the Assignment and Transportation Problems. *SIAM Journal on Computing*, 5(1):32–38, 1957.

[Nic90] J. R. Nickolls. The Deisgn of the MasPar MP-1. In *Proceedings of the 35th IEEE International Computer Conference*, pages 25–28, March 1990. San Francisco.

[Pos80] J. B. Postel. User Datagram Protocol. In *Internet Request for Comments RFC-768*, 1980.

[Ran87] A. G. Ranade. How to emulate shared memory. In *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1987.

[Red78] S. F. Reddaway. DAP- A Flexible Number Cruncher. In *Proceedings of the LASL Workshop on Vector and Parallel Processors*, 1978. Los Alamos.

[RJN90] R. Raghavan, K. K. Jung, and H.T. Nguyen. Fine Grain Parallel Processors and Real-Time Applications: MIMD Controller/SIMD Array. In *Proceedings of 10th Inetrnational Conference on Pattern Recognition*, pages 324–331, June 1990. Atlantic City, New Jersey.

[Sco80] F. Van Scoy. The Parallel Recognition of Classes of Graphs. *IEEE Tranactions on Computers*, 29(7):563–570, July 1980.

[Sub92] R. Subramonian. Designing synchronous algorithms for asynchronous processors. Technical report, University of Californinia, Berkeley, 1992.

[Sun91] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. Technical report, Emory Unversity and Oak Ridge National Laboratory, 1991.

[Sun92] V. S. Sunderam, February 1992. Personal Communication.

[WZ91] J. M. Wein and S. A. Zenios. On the Massively Parallel Solution of the Assignment Problem. *Journal of Parallel and Distributed Computing*, 13:228–236, 1991.