Experiments in Parallel Scene Labeling

Monroe Thomas[1] and Xiaobo Li[2]

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1
(403) 492-2299, Fax: (403) 492-1071, li@cs.UAlberta.CA

Abstract

This paper reports several experiments in object labeling, especially applied to scene analysis, which expands our previous results [2]. We first investigate the implementation of efficient sequential labeling algorithms. An array indexing method is employed to utilize filtering before the tree search. The more compact data structure and the filtering process significantly improve the sequential program (by an order of magnitude), and thus provide a much more meaningful ground for evaluating the speedup of parallel methods. We then develop a parallel labeling algorithm on a Myrias SPS-2 machine with 64 processors. The reason for choosing this particular machine is the simplicity of its MIMD architecture: other, more complex architectures should be able to improve on the techniques presented here. The algorithms employ fixed-depth, fixed-breadth, and breadth-first approaches when processing the search tree to take advantage of the parallel computation resource. Two new object ordering heuristics are introduced. Besides the line drawing labeling (straight line interpretation) problems, some edge orientation problems are also tested, in which much larger trees are involved. The experimental results reveal the complex nature of the parallel labeling problem and some key issues in implementation. Reasonable speedups have been achieved on the SPS-2, using the new ordering and filtering techniques proposed in this paper.

Keywords constraint satisfaction, scene analysis, edge orientation, tree search, speedup, MIMD

1. Introduction

The consistent labeling problem has attracted great attention in computer vision and artificial intelligence areas. We define the consistent labeling problem following the popular notation in the literature. Set U contains m units (objects), $U = \{u_1, \ldots, u_m\}$. Set Λ contains l labels, $\Lambda = \{\lambda_1, \ldots, \lambda_l\}$. Let R be a unit-label constraint relation: $R \in (U \times \Lambda)^n$. A (consistent) labeling of the units in U is a function $f : U \rightarrow \Lambda$ such that for all $(u, v) \in U \times U$, $(u, f(u), v, f(v)) \in R$.

The general consistent labeling problem is denoted by CLP, while some special cases are given names as CLDP, CLEP, CLSP, and CLGP [10]. In a consistent labeling *decision* problem (CLDP), the existence of a solution (consistent labeling of the units) is decided. In an consistent labeling *enumeration* problem (CLEP), the number of solutions is the desired result. In a *search* (CLSP) problem, only one solution is needed. In a *generation* problem (CLGP), all possible solutions are to be found. Different techniques are used for different CLP types. In our study, the CPGP problem and CPSP problems are investigated.

Two basic approaches to consistent labeling are known as tree search and filtering [10]. Tree search is guaranteed to find all solutions if there are any. Various filtering algorithms have been applied to reduce the possible labels each unit could take and to simplify the constraint relation set R. These filtering algorithms are used before and during the tree search to eliminate unnecessary comparisons. Waltz [12] proposed a filtering algorithm which is aimed at reducing the set of possible labels that each unit is allowed to take. This filtering, classified as *domain-filtering*, can be implemented as a preprocessing step before the tree search or utilized during the tree search, and has proved to be very efficient in the line interpretation application. Montanari's approach [9] eliminates unacceptable constraint tuples, i.e., reduces the R set. This filtering technique is classified as *relation-filtering*. Relation-filtering is not cost-effective when used during the tree search [10]. Many researchers have devoted their effort to developing an efficient algorithm which combines filtering (mostly domain-filtering) and tree search techniques [10]. The balance of the the contribution of the two components may be quite different for different types of CLP applications. Haralick and Elliott [4] tested several hybrid CLP algorithms on N-queens problems and some randomly generated problems where $n = 2$ (binary constraint). It is concluded from [4] that *bit-parallel forward checking* method performs better than other methods which use more filtering. Nudal [10] carried out expected-complexity analyses of the computation complexity of different CLP algorithms. In our experiments, CLP problems with $n = 2$ and $n = 3$ are considered. We employ a bit-parallel tree search sequential algorithm with domain and relation filtering during the preprocessing step.

Several researchers proposed parallel algorithms and multiple processor architectures for solving CLP problems ([1], [3], [6], [7], [11]). Most of the research in this area concentrates on SIMD approaches. McCall *et al* [8] con-

[1]Monroe Thomas is supported in part by the Canadian Natural Sciences and Engineering Research Council under Grant 58-01024-1540 and Grant OGP9198

[2]Xiaobo Li is supported in part by the Canadian Natural Sciences and Engineering Research Council under Grant OGP9198. All correspondence should be addressed to Xiaobo Li.

ducted a simulation study on an eight processor MIMD system for problem solving strategies for CLP with binary constraints. The computer architecture suggested in [8] is an interrupt system. Gregorish *et al* [2] investigated the behavior of parallel MIMD CLP algorithms on a Myrias SPS-2 computer systems, which has 64 processors and no inter-task communication. Line interpretation in a straight line scenes were considered, and two unit ordering heuristics were studied in detail. In the current paper, we report some results of our further experiments and discuss several implementation issues. With more compact data structure and domain/relational filtering, the sequential algorithm has been significantly improved, thus providing a more meaningful ground for evaluating the speedup of parallel methods. Other unit ordering methods and their effect to both sequential and parallel CLP, the use of variable search depth, and the option of breadth-first search techniques are investigated. Many researchers so far concentrated on CLP problems with binary constraints and used line interpretations for testing purposes. To be more general, we tested our parallel methods on CLP problems with $n = 2$ and $n = 3$, and included some "edge-orientation" [5] problems.

In Section 2, we describe a sequential implementation of the CLP algorithm and report experimental results. The experiments on the Myrias SPS-2 are reported in Section 3. A brief conclusion is given in Section 4.

2. Sequential Implementation of CLP Algorithms

The procedure of the sequential depth-first tree search algorithm is as follows:

```
Procedure descend(L, u)
labeling L;
object u;
{
    for (each label for object u) {
        assign label to object u in labeling L;
        if (label violates any constraints)
            next label;
        else if (u is the last object)
            print labeling L;
        else
            descend(L, u + 1);
    }
    return;
}
```

We implemented this depth-first tree search with filtering and ordering preprocessing.

2.1 Data Structure and Filtering

In our implementation of the sequential CLP algorithm, a binary vector is used to indicate which labels are permissible for each unit. This method is called *bit parallel* by Haralick *et al* [4]. Corresponding to each unit u_i, a binary number (word) W_i is established with bit k relating to the k-th label. That is, $W_i[k] = 1$ indicates that unit u_i could take label λ_k. An array RSET is used to store and manipulate the constraint set R. Each element of this array corresponds to a unit-tuple which is involved in R. All possible labelings of this tuple are stored in a linked list attached to this RSET element. The size of this tuple (the number of units in this tuple) is also stored (for implementation purposes).

In the following discussion, we will consider an example with six units and two labels:

$U = \{0,1,2,3,4,5\}$,
$\Lambda = \{\lambda_0 = a, \lambda_1 = b\}$,
$R = \{(0,a,1,a,2,a),(0,a,1,b,2,b),(0,a,1,b,2,a),(0,a,1,a,2,b),$
$(0,b,1,a,3,b),(0,a,1,a,3,b),(0,a,1,b,3,b),(0,b,1,b,3,b),$
$(1,a,2,b,4,a),(1,a,2,b,4,b),(1,b,2,a,4,a),(1,b,2,a,4,b),$
$(1,a,3,a,4,a),(1,a,3,b,4,b),(1,b,3,a,4,b),(1,b,3,b,4,b),$
$(4,a,5,a),(4,b,5,a),\}$

The binary numbers indicating permissible labels for the units initially are:

$W_0 = 11$, $W_1 = 11$, $W_2 = 11$, $W_3 = 11$, $W_4 = 11$, and $W_5 = 11$.

The bit vector 11 indicates labels 'a' and 'b' are permissible, a bit vector 01 means only label 'a' is valid.

The array RSET has the following five elements:
($\rightarrow$ denotes a linked list)

index	tuple-size	unit tuple	possible labeling of the tuple
0	3	(0, 1, 2)	$(a,a,a) \rightarrow (a,b,b) \rightarrow (a,b,a) \rightarrow (a,a,b)$
1	3	(0, 1, 3)	$(b,a,b) \rightarrow (a,a,b) \rightarrow (a,b,b) - (b,b,b)$
2	3	(1, 2, 4)	$(a,b,a) \rightarrow (a,b,b) \rightarrow (b,a,a) - (b,a,b)$
3	3	(1, 3, 4)	$(a,a,a) \rightarrow (a,b,b) \rightarrow (b,a,b) - (b,b,b)$
4	2	(4, 5)	$(a,a) \rightarrow (b,a)$

Any unnecessary unit-label constraints can be eliminated from RSET in a process known as relational filtering. For example, the constraints defined for the tuple (0,1,3) preclude unit 3 from ever having the label 'a'. Thus the constraints (a,a,b) and (b,a,b) for the tuple (1, 3, 4) can be removed, which then prevents unit 4 from having label 'a'. Similarly, unit 5 cannot ever have the label 'b'. This process is applied through the RSET structure several times until no more constraints can be removed. The above example will produce the following reduced RSET:

index	tuple-size	unit tuple	possible labeling of the tuple
0	3	(0, 1, 2)	$(a,a,a) \rightarrow (a,b,b) \rightarrow (a,b,a) - (a,a,b)$
1	3	(0, 1, 3)	$(a,a,b) \rightarrow (a,b,b)$
2	3	(1, 2, 4)	$(a,b,b) \rightarrow (b,a,b)$
3	3	(1, 3, 4)	$(a,b,b) \rightarrow (b,b,b)$
4	2	(4, 5)	(b,a)

The W words are now modified accordingly and become: $W_0 = 01$, $W_1 = 11$, $W_2 = 11$, $W_3 = 10$, $W_4 = 10$, and $W_5 = 01$.
Reducing the possible labels each unit can take is known as domain filtering.

In our experiments, we tested three filtering options:

- Filter 0: No filtering applied;
- Filter 1: Domain filtering applied;
- Filter 2: Domain and relation filtering applied.

Domain filtering alone is not as effective as when combined with relational filtering. In the above example, we can assign $W_5 = 01$ without removing any tuples from the constraint set, because no constraint ever lets unit 5 have label '*b*'. We could not assign $W_3 = 10$ based on the the constraints for tuple (0, 1, 3) alone, since tuple (1, 3, 4) has constraints containing possible labels for unit 3 which are '*a*'. Relational filtering allows us to remove all RSET constraints associated with (1, 3, 4) where unit 3 is assigned '*a*'. Only then can we decide $W_3 = 10$.

2.2 Unit Ordering

Gregorish *et al* [2] reported that the performance of CLP is strongly tied to the order in which the units are placed in the tree search. It is beneficial to place the units with most constraints at the top of the tree (close to the root), and place the units with less constraints at the bottom of the tree (close to the leaves). By reducing the amount of branching at the upper part of the tree, one can minimize the effort of backtracking. For the above example (after domain and relational filtering have been applied), the search order of 0, 3, 4, 5, 1, 2 has a depth-first search path length of 16, while the order 0, 1, 2, 3, 4, 5 has a path length of 34.

In the past, the choice of different ordering heuristics depended heavily on the the type of problem. These heuristics, such as the spatial relation heuristic discussed in [2] can require complex data processing and some user interaction. It would be beneficial to develop an application independent ordering method that can do as well as any application dependent heuristic. It should be noted that the time saving realized by intelligent ordering of the units has to be balanced with the computation effort of the ordering itself. In our experiments, several unit ordering methods are tested, all application independent.

- Order 0: no ordering used
- Order 1: the unit-constraint ordering heuristic used in [2]. This method was reported to perform nearly as well as manually ordering the objects for optimal results. Note that this ordering was originally developed for non-filtering search tree algorithms. We test it to see if its performance is still good when used with filtering.
- Order 2: the units are ordered according to the numbers of times each unit appears in the set R. The unit that appears more often than others will be searched first, *i.e.*, at the root of the tree. The (naive) assumption made is that the frequency of a unit's appearance is proportional to how constrained it is with respect to the other units.
- Order 3: the units are ordered according to the numbers of possible labels they can take. The unit with the least number of possible labels is placed first, since it has the lowest branching factor. This method was developed as a consequence of domain filtering.

Orderings 1 and 2 depend only upon unit-unit relationships, they do not account for the unit-label relationships. Order 1 requires a recursive, set splitting algorithm that can require $O(n^2)$ time to complete. Order 2 is easy to implement with a quickersort algorithm to sort units in descending order by number of appearances in RSET. Order 3 looks at the unit-label relationships as well as unit-unit relationships. This method minimizes the amount of backtracking needed while performing depth-first searches. All the work needed for this method is done by the domain filtering process, so we just use a quickersort algorithm to sort the units in ascending order by the number of bits in the corresponding binary words (bit vectors).

2.3 Array Indexing

For the sake of clarity we reproduce the reduced RSET table from section 2.1:

index	tuple-size	unit tuple	possible labeling of the tuple
0	3	(0, 1, 2)	$(a,a,a) \rightarrow (a,b,b) \rightarrow (a,b,a) \rightarrow (a,a,b)$
1	3	(0, 1, 3)	$(a,a,b) \rightarrow (a,b,b)$
2	3	(1, 2, 4)	$(a,b,b) \rightarrow (b,a,b)$
3	3	(1, 3, 4)	$(a,b,b) \rightarrow (b,b,b)$
4	2	(4, 5)	(b,a)

We use a parameter P for each unit to help decide how we are going to make comparisons between the constraint sets and the search tree. A unit with a low P value occurs in the search tree before units with high P values. In the above example, if the units are ordered as 0, 3, 4, 5, 1, 2, unit 0 has the lowest P value, and unit 2 has the highest. The P values are assigned after the units are ordered (by whatever method).

At each level of the tree, starting from the root, RSET is examined to see if any RSET element has its unit tuple currently appearing in the tree. Consider again the example given above, and assume that the units are searched in the order of 0, 3, 4, 5, 1, 2. At the second level (unit 3), no RSET tuples need to be examined since all unit-unit constraints involving unit 3 contain some units with higher P value. These units have not been reached in the tree yet and it is useless to consider them. At level 4 (unit 5), we need only consider the RSET element associated with the unit-unit constraint (4, 5), since unit 4 is in the search tree. At level 5 (unit 1), the RSET elements corresponding to unit tuples (1, 3, 4) and (0, 1, 3) need to be examined.

We would like an efficient method of directly accessing only the constraint sets in RSET that we need to. Since sorting the linked lists in RSET to match the search order would be too inefficient, and sacrificing the flexibility of linked lists is undesirable, we introduce the array R_INDEX to facilitate this direct access to the applicable RSET tuples for any given depth in the search tree. R_INDEX has the same number of elements as RSET.

Each R_INDEX element corresponds to an element in RSET, and contains three fields: Hi_P, Lo_P, and Index_in_RSET. Hi_P is the unit with the highest P value involved in this R tuple, and Lo_P is the unit in this tuple that has the lowest P value. Index_in_RSET links this R_INDEX element to an RSET element. For the above example, the R_INDEX can be given below.

index	Hi_P	Lo_P	Index_in_RSET
0	5 ($P = 3$)	4	4
1	1 ($P = 4$)	0 ($P = 0$)	1
2	1 ($P = 4$)	3 ($P = 1$)	3
3	2 ($P = 5$)	0 ($P = 0$)	0
4	2 ($P = 5$)	4 ($P = 2$)	2

The elements of this array were sorted by the Hi_P field in ascending P value order, thus in the order that these elements (and their corresponding RSET elements) are involved in the tree search. When there is a tie in Hi_P field, the ascending order of Lo_P field is used.

When we start the tree search, we do not have to make a constraint set comparison until we reach level 4 (unit 5), and then, by reference to R_INDEX, we note that unit 5 requires a comparison with the constraints at index 4 of RSET. The next comparison will not have to be made until level 5 (unit 1), and then again, by reference to R_INDEX, we have direct and immediate access to the appropriate constraint sets in RSET.

2.4 Sequential Experimental Results

In this experiment, eight data sets are used. Table 1 gives some description of the data sets.

Table 1 Test Cases

case id	number of objects	size of tree	number of solutions
9	9	4^9	4
20(#1)	20	5^{20}	1
20(#2)	20	4^{20}	10
33	33	4^{33}	120
54	54	4^{54}	75
80	80	4^{80}	25,920
256(#1)	256	8^{256}	13,271,000
256(#2)	256	8^{256}	1,650,458,624

The first six sets are line interpretation problems with 9 to 80 units, including two problems with 20 units. Case 20(#1) is identical to Case 20(#2), except another label has been used to further constrain the set so only one solution is possible. All of these data sets were used in [2]. The last two data sets with 256 units are edge orientation problems [5], illustrated in Fig. 1. The possible orientations of each edge pixel are marked. The purpose of labeling is to determine the most "reasonable" orientation for each edge.

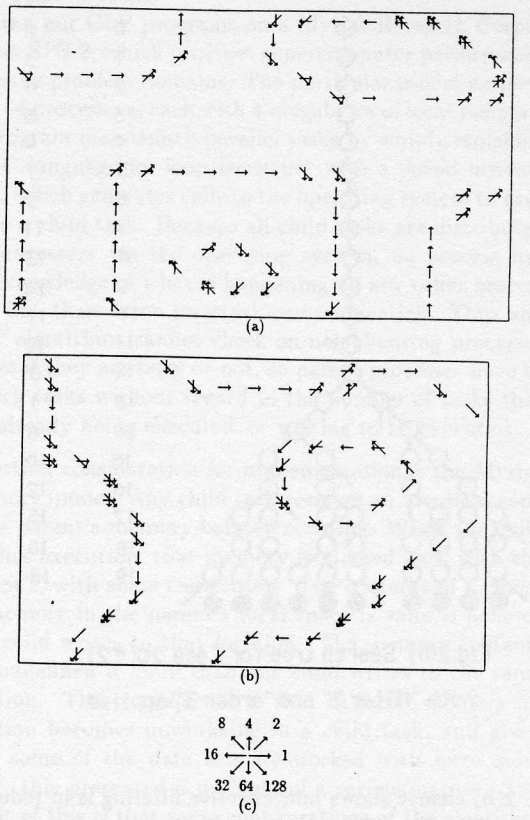


Fig.1 Possible orientations of each edge pixel in the cases with 256 units (a) Case 256(#1), (b) Case 256(#2), (c) the label for each direction.

Fig. 2 shows the difference filtering and ordering can make on the shape of the search tree.

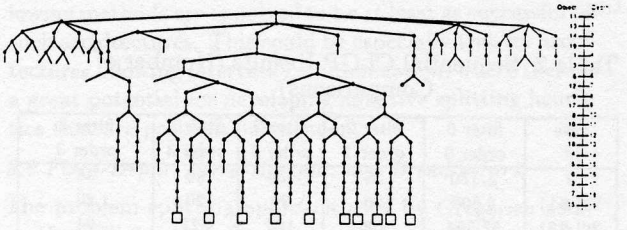
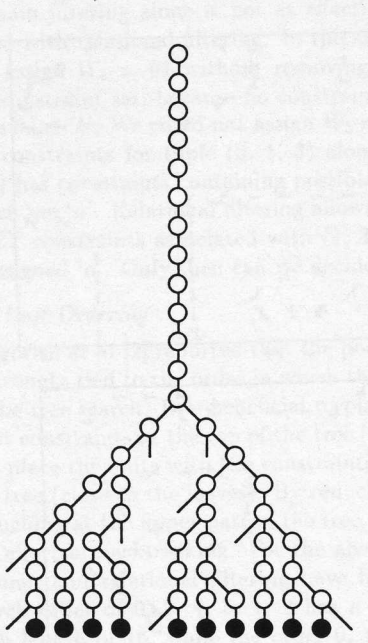


Fig.2(a) Search tree for Case 20(#2) with "filter 0" and "order 1" applied



depth	unit
0	2
1	3
2	4
3	5
4	6
5	7
6	9
7	11
8	13
9	14
10	17
11	19
12	0
13	1
14	8
15	10
16	12
17	15
18	16
19	18

Fig.2(b) Search tree for Case 20(#2) with "filter 2" and "order 3" applied

Fig. 2(b) clearly shows how effective filtering is in reducing the branching factor at each level of the tree. Order 3 places the units so that a slim, balanced search tree is obtained, and solutions can be found rapidly in an orderly manner. The order 1 method illustrated in Fig. 2(a) can be seen to have many redundancies in its search tree.

Table 2 shows the number of comparisons (of the search tree to the constraint set) required to solve the CLGP using various combinations of filtering and ordering heuristics.

Table 2 Sequential CLGP Results (Number of Comparisons)

case id	filter 0 order 0	filter 2 order 1	filter 2 order 2	filter 1 order 3	filter 2 order 3
9	2,180	35	136	80	35
20(#1)	8,600	20	20	20	20
20(#2)	67,364	206	196	322	94
33	7,208,468	3,204	2,482	899	549
54	—	24,495	30,718	6,850	720
80	—	1,666,649	596,754	201,912	124,318
256(#1)	—	—	—	—	29,689,800
256(#2)	—	—	—	—	3,700,428,112

filter 0: no filtering is applied,
filter 1: only domain filtering is applied,
filter 2: both domain and relation filterings are applied;
order 0: units are searched in a random order,
order 1: units are searched according to ordering method 1.
order 2: units are searched according to ordering method 2,
order 3: units are searched according to ordering method 3,
—: the program does not finish in 10 days

Each comparison takes an average of 0.18 ms on a Sun 3/50 using the array indexing method of accessing the constraint set. It is interesting to note that case 256(#2) took 7.7 days to find all solutions. It is also interesting to see how many more comparisons are required by orders 1 and 2 than by order 3. The obvious explanation is the reduction in the amount of backtracking in the tree by order 3, and thus the redundancies encountered when using orders 1 and 2.

Fig. 3 graphically illustrates the relationship between different ordering and filtering combinations.

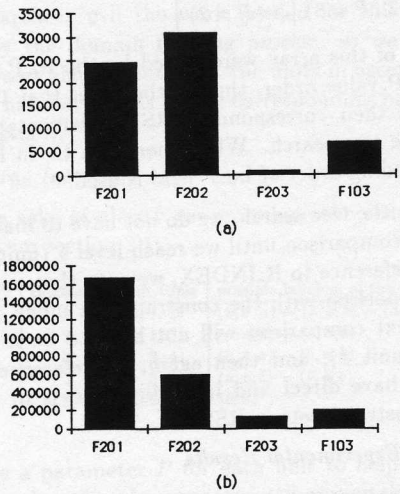


Fig.3 Sequential CLGP results (number of comparisons)
(a) case 54. (b) case 80.
F201: filter 2, order 1; F202: filter 2, order 2;
F203: filter 2, order 3; F103: filter 1, order 3.

The conclusion we draw from Fig. 3 is that filter 2 (domain and relation filtering) and order 3 (minimal backtracking) are superior to other filterings and orderings. This is not surprising, as the combination of domain and relation filtering can greatly reduce the constraint set, while minimal backtracking reduces the amount of time spent in the tree.

Table 3 shows the number of comparisons required between the search tree and constraint set for finding the first solution in the search tree (CLSP).

Table 3 Sequential CLSP Results (Number of Comparisons)

case id	filter 1 order 1	filter 1 order 3	filter 2 order 1	filter 2 order 3
9	46	46	15	15
20(#1)	20	20	20	20
20(#2)	1,736	190	28	22
33	40,138	280	59	37
54	3,864,940	1,774	651	66
80	—	42,126	102,257	89
256(#1)	261	259	256	256
256(#2)	257	257	257	257

A variety of combinations of filtering and ordering heuristics were used when solving CLSP in a process called the "Eureka jump". The "Eureka jump" simply stops the tree search as soon as the first solution is found. The last column in Table 3 again illustrates the effectiveness of domain and relation filtering with minimal backtracking (Fig. 4). We are pleasantly surprised to note that this highly efficient and powerful combination allows us to find the first solution with a number of comparisons approaching the order of the number of units themselves!

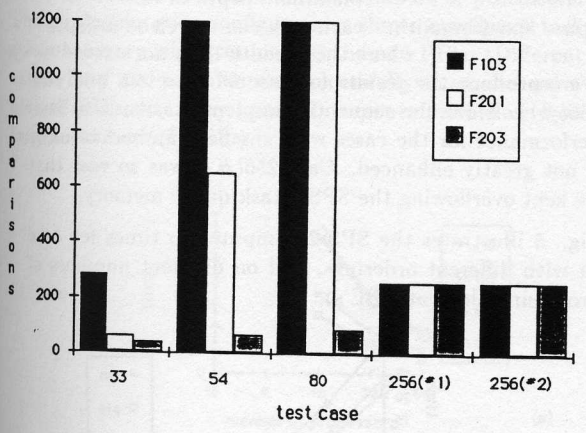


Fig.4 Sequential CLSP results (number of comparisons)

F103: filter 1, order 3; F201: filter 2, order 1; F203: filter 2, order 3

In summary, the development of a highly efficient sequential CLGP algorithm is the first step in creating efficient parallel solutions. It provides a much more meaningful ground for the study of parallel processing behaviour, especially when considering speedups. The most efficient sequential implementation we tried incorporated domain and relation filtering while ordering the units to provide minimal backtracking in the tree. The two heuristics go hand in hand: the reduction in the number of possible labels by domain and relation filtering allows us to order the units such that the tree is slim and balanced, which leads to an orderly discovery of solutions.

3. Parallel Implementation of CLP Algorithms

MIMD approaches to CLP are generally of a divide-and-conquer nature, due to the multiple instruction nature of the architecture. We can split a big problem into smaller subproblems, which are solved in parallel to produce a final solution. Since each branch of a CLP search tree is independent of any others, an MIMD approach involving some method of problem splitting seems to be an ideal way to improve execution time over the sequential method discussed in section 2.

3.1 Myrias SPS-2

We ran our CLP programs on a Myrias Research Corporation SPS-2, which provides supercomputer performance in many problem domains. The particular model we used had 64 processors, each with 4 megabytes of local memory. A program may launch parallel tasks by simply replacing the C language for-loop construct with a *pardo* instruction, which generates calls to the operating system to generate a child task. Because all child tasks are distributed to processors via the operating system, no process has any knowledge of what is happening on any other processor, *i.e.*, there is no intertask communication. Thus any CLP algorithms cannot check on neighbouring processes to see if they are busy or not, so parent processes have to launch tasks without regard to the number of tasks that are already being executed, or waiting to be executed.

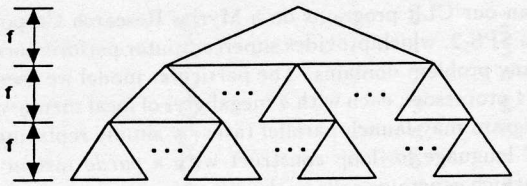
A further consideration for implementation is the Myrias memory model. Any child task receives an identical copy of its parent's memory before executing. When the child finishes execution, that memory is merged back with the parent's, with some restrictions. The contents of a chunk of memory in the parent's local space is valid if none or one child writes to that location. The memory contents are undefined if more than one child writes to the same location. The consequence is that dynamic memory allocation becomes unworkable in a child task, and given that some of the data sets we worked with were quite large, this presented a problem of a serious nature. The result of this is that some configurations of the algorithm crashed on the SPS-2. In general though, we were able to run much larger problems than Gregorish *et al*, who used the same machine.

All problem splitting approaches had to be implemented with no intertask communication, and with attention paid to limiting the amount of waiting tasks that had to be queued. Most other MIMD architectures do not suffer from these constraints, so any implementation of the following methods are expected to be at least as successful on such architectures. This could be especially true for architectures allowing intertask communication, where there is a great potential for developing adaptive splitting heuristics that are problem independent.

3.2 Fixed-Depth, Fixed-Breadth, and Breadth-First

The problem splitting approach used by Gregorish *et al* on the SPS-2 was to fix the depth that a particular process could search to, and then all valid partial labelings generated by that process would be launched as parallel tasks. We investigate this fixed-depth approach, and also examine some alternatives: fixed-breadth, and breadth-first.

The fixed depth scheme can be illustrated as follows:

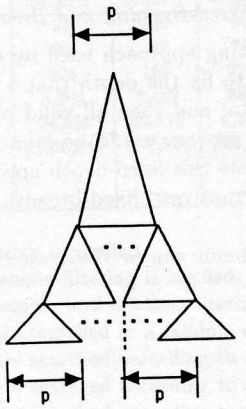


The depth f of any child task is fixed upon execution of the program, and is determined by the user. Note that if f = # of units, then the problem is solved purely sequentially. If $f = 1$, then the method is purely parallel. Gregorish *et al* varied f for many data sets, and found that a fixed depth of 6 generally performed well in all cases [2]. Values for f greater than 6 inhibited the parallelism, and values less than 6 generated too much overhead when distributing parallel tasks. The value for f was empirically determined, there was no method proposed for determining f heuristically.

The fixed-breadth and breath-first splitting schemes were designed to get some control over how many parallel tasks were launched by any one process. We felt that some knowledge of what other processes were doing would be beneficial for another process to decide whether to launch a task (if there was an idle processor). Since this information is not available on the SPS-2, there was no way of knowing how many tasks were being launched by each process.

The idea behind the breadth-first scheme was to generate as many tasks as possible at each depth. If more tasks were launched than processors available, the SPS-2 operating system would queue tasks up to be executed when processors became available. The danger of this method is overloading the queue memory with a large number of waiting tasks. We theorized that the advantage to this method would be little idle processor time.

Fixed-breadth allowed us to establish an upper limit on the number of tasks launched by each process. A fixed-breadth search tree is illustrated below:



The breadth p to which a process may carry its sequential search of the tree is given to the program at run-time, and we varied p over a wide range to see if there is any one value of p which works well for most cases. Note that a value of $p = 1$ results in a purely parallel execution, while a sufficiently large p will cause purely sequential operation. Fixed-breadth is no more complex to implement than fixed-depth, since our method already keeps track of valid labels for each unit (via domain and relation filtering). Once an ordering is decided upon, we then know the branching factor at each depth in the tree.

3.3 Parallel Experimental Results

The notation for the experimental results:

- f*n*: fixed-depth: each task can search to a fixed maximum depth of *n*
- p*n*: fixed-breadth: each task can search to a depth where the breadth does not exceed *n*

We reproduce the results for case 54, case 80, and case 256(#1). Since the sequential implementation is so fast, performance for the cases with smaller numbers of units is not greatly enhanced. Case 256(#2) was so vast that we kept overflowing the SPS-2 task queue memory.

Fig. 5 illustrates the SPS-2 computation times for case 54 with different orderings, and on different numbers of processing elements (PE's).

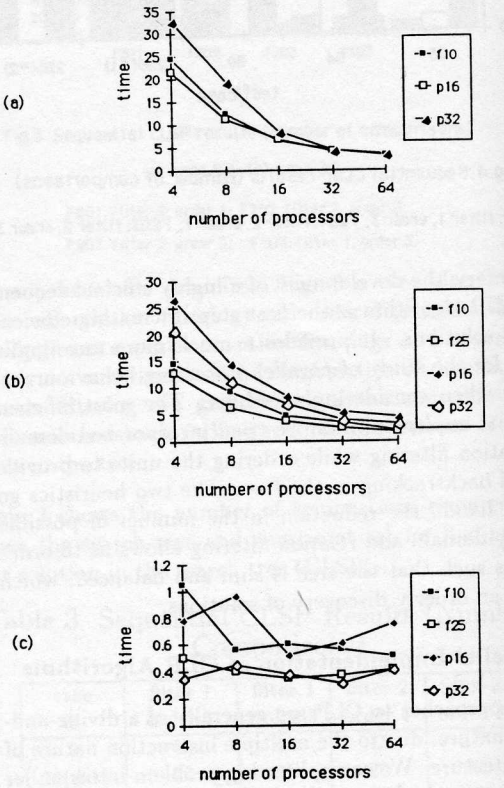


Fig. 5 Parallel computation time (sec.) for Case 54

(a) order 1, (b) order 2, (c) order 3.

Fig. 5(a) represents the Gregorish *et al* unit-ordering heuristic (order 1). It overflowed task queue memory when running $f25$. Order 1 and order 2 produced similar results, and the execution times are slower than for sequential. When compared with Fig. 5(c), which shows times just as fast or faster than sequential, we note that orders 1 and 2 have much more backtracking in their search trees than order 3. We conclude this redundancy caused by backtracking leads to inefficient utilization of processors, *i.e.*, unnecessary task generation through repetition of work. Another conclusion we draw from Fig. 5(c) is that case 54 is not really large enough for all 64 PE's to get involved, thus 4 PE's performs just as well.

Fig. 6 illustrates computation times for case 80 with different orderings, and on different numbers of processors. The implementation configurations shown overflowed task queue memory when run on less than 16 PE's. (Different values for f and p can be run successfully on smaller numbers of PE's, but they generally execute more slowly on larger numbers of PE's.)

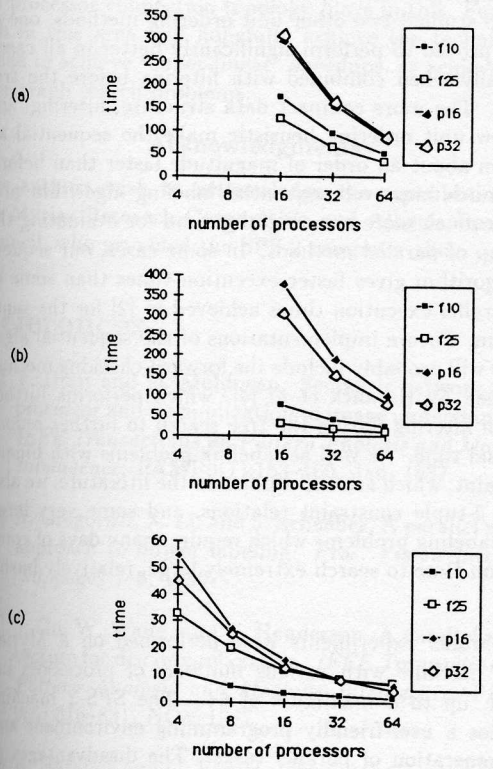


Fig. 6 Parallel computation time (sec.) for Case 80

(a) order 1, (b) order 2, (c) order 3.

Again, Fig 6(a) and (b) show that there are too many redundant tasks being created as a result of the unbalanced nature of the trees created by order 1 and 2. These execution times are still slower than for the sequential case, but the fixed-depth problem splitting scheme appears to be better than the fixed-breadth. Fig. 6(c) shows the benefits of parallelism when using order 3: all methods at 16 PE's finished faster than the sequential case. Method $f10$ always finished faster than the sequential case (22 seconds), in fact, method $f10$ on 64 PE's finished in 1.43 seconds. Another test, $f64$ (not shown), managed to finish in 1.26 seconds.

Fig. 7 illustrates the times for finding all solutions for case 256(#1) using order 3. Methods $p16$ and $p32$ timed out before completion (and so are not shown), but $p64$ provided fairly good results.

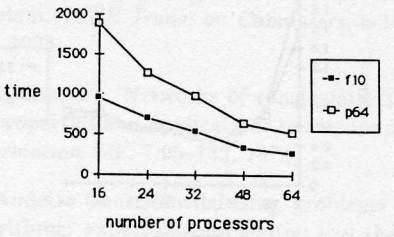


Fig. 7 Parallel computation time (sec.) for Case 256(#1)

(order 3 is applied)

Method $f10$ solved case 256(#1) in about 3.25 minutes, which is much better than the 89 minutes the sequential solution took. In fact, a fixed-depth of 10 gave the best results overall. Unfortunately, there does not seem to be very much correlation between any certain fixed-breadth value and good performance for most cases.

Fig. 8 shows speedup curves for case 80 and case 256(#1) on different numbers of processors using order 3 and different splitting methods.

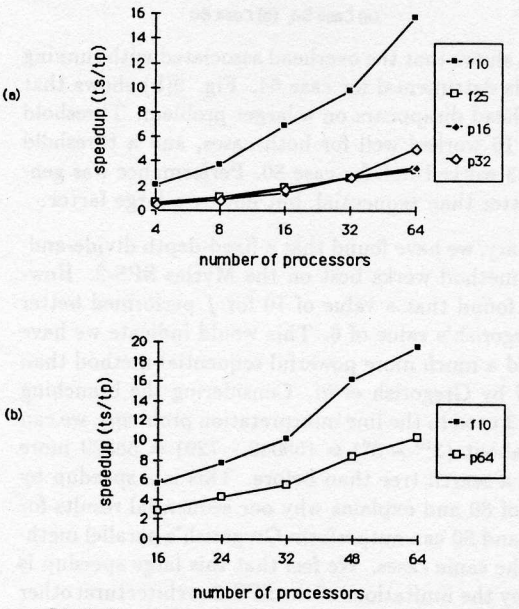


Fig.8 Speedup comparison for Case 80 and Case 256(#1)

(a) Case 80, (b) Case 256(#1)

Gregorish *et al* suggest that speedups increase with larger problem sizes. We agree with this; we achieved a speedup factor of 16 with $f10$ on case 80 and a speedup factor of about 20 with $f10$ on case 256(#1). For smaller numbers of units, parallel performance tails off due to the efficiency of the sequential algorithm and the overhead associated with launching parallel tasks.

Fig. 9 shows the results of the breadth-first searches on 32 and 64 PE's. Only cases 54 and 80 are illustrated for different threshold levels. The threshold level is the point at which there are enough partial labelings to merit launching parallel tasks.

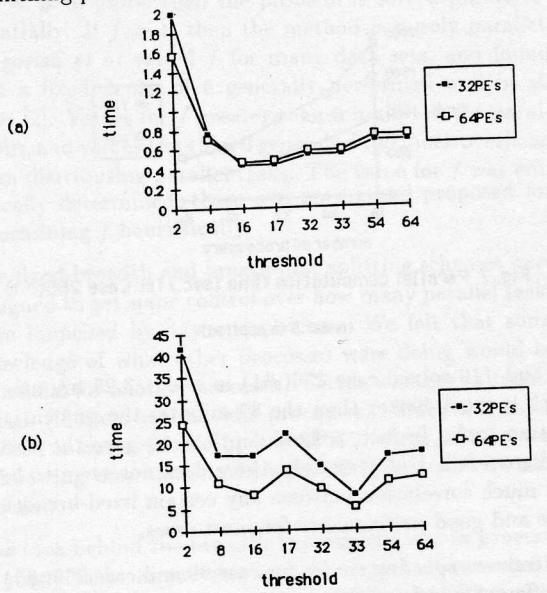


Fig. 9 Computation time (sec.) of breadth-first search method

(a) Case 54, (b) Case 80

Fig. 9(a) shows that the overhead associated with running 64 PE's is detrimental for case 54. Fig. 9(b) shows that this overhead disappears on a larger problem. Threshold levels of 16 worked well for both cases, and a threshold level of 33 worked well for case 80. Performance was generally better than sequential, but not by a large factor.

In summary, we have found that a fixed-depth divide-and-conquer method works best on the Myrias SPS-2. However, we found that a value of 10 for f performed better than Gregorish's value of 6. This would indicate we have developed a much more powerful sequential method than was used by Gregorish *et al*. Considering the branching factor of 3 used in the line interpretation problems, we can process about $(3^{10} - 3^6) = (59049 - 729) = 58320$ more nodes in a search tree than before. This is a speedup by a factor of 80 and explains why our sequential results for cases 54 and 80 can outperform Gregorish's parallel methods for the same cases. We feel that this large speedup is masked by the limitations of the SPS-2 architecture; other MIMD architectures with memory sharing and intertask communication can eliminate some of the overhead associated with the task queuing and memory merging that the SPS-2 requires. We also found that Gregorish's ordering method combined with filtering can lead to search trees that contain much redundancy. This leads to inefficient use of processors in a parallel environment, since there is much unnecessary duplication of effort. Order 3 minimized this redundancy and led to efficient tree searches. We did not perform eureka jump testing on the SPS-2 for two reasons: the sequential algorithm is is extremely efficient at this (see F2O3, Fig. 4), and implementation also becomes difficult without intertask communication.

4. Conclusions

This paper reports extensive experiments in consistent labeling algorithms applied to scene object labeling. An implementation of fast sequential algorithms and several issues involved in different types of labeling problems are investigated. A bit-parallel labeling representation and data structure speeds up constraint set comparisons during the tree search and different filtering algorithms are used in the search to reduce the size of the constraint set and label domain. Along with a unit-constraint object ordering heuristic proposed in our previous paper [2], we also studied two other unit ordering methods, one of which proved to perform significantly better in all cases, especially when combined with filtering before the tree search. The more compact data structure, filtering, and the new unit ordering heuristic make the sequential algorithm about an order of magnitude faster than before. This much improved sequential labeling algorithm provides a much more meaningful ground for evaluating the speedup of parallel methods. In some cases, our sequential algorithm gives faster execution times than some of the parallel execution times achieved in [2] for the same problem. Future implementations of the sequential algorithms will probably include the forward-checking method discussed by Haralick *et al* [4], which performs further domain filtering during the tree search to further reduce traversal time. As well as labeling problems with binary constraint, which are well studied in the literature, we also tested 3-tuple constraint relations, and some very large scene labeling problems which require many days of computation time to search extremely large, relatively bushy trees.

Our parallel experiments were performed on a Myrias SPS-2 machine with various numbers of processors involved, up to a maximum of 64. The SPS-2 machine provides a user-friendly programming environment and easy generation of parallel tasks. The disadvantages of using the SPS-2 are no communication between processes and no dynamic memory allocation within child processes. Our tests employed this machine as a basic MIMD model to reveal some of the inherent parallelism of the labeling problem and to investigate several issues involved in implementing parallel labeling methods. The results of our experiments on this simple MIMD architecture shows that depth-first parallel search with fixed-depth divide-and-conquer strategies, working with filtering and our optimal unit ordering algorithm, achieves reasonable speedups.

The techniques which proved possible and effective on the SPS-2 should be useful and very easy to implement on other, more complex MIMD architectures. Such MIMD architectures should be able to exploit the increased performance of the sequential algorithm to a much greater extent than the SPS-2, especially in CLSP applications. Our study stresses the importance and the special nature of inter-processor communication for the labeling problem, and provides insights into future research in this area.

We applied our parallel algorithms (with modification) on a distributed environment, *i.e.*, a network of Sun workstations. The inter-processor communication facility is attractive when implementing "Eureka jump" methods for CLSP applications. Our initial attempts were based on a client/server process model for handling partial labelings and passing messages. However, the speedups provided by eureka jumps were not as impressive as we expected, perhaps due to the efficiency of the sequential algorithm, and limitations in the speed and design of our partial labeling "server". This raised questions about moving large numbers of partial labelings between processes, and the role that processor connection topology plays in this. Further work in this area will hopefully achieve the techniques needed to achieve "super-linear" speedups, as achieved in other graph search problems.

Acknowledgments

The authors wish to acknowledge the contributions of the Myrias Research Corporation and Alberta Research Council, who provided the SPS-2 computer time.

References

[1] V. Dixit and D. Moldovan. Semantic network array processor and its application to image understanding. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(1):153–160, Jan. 1987.

[2] R. Gregorish, X. Li, and J. Schaeffer. A parallel mimd approach to object labeling. *Proc. Vision Interface '90*, pages 1–8, 1990.

[3] J. Gu, W. Wang, and T. Henderson. A parallel architecture for discrete relaxation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(6):816–831, NOV 1987.

[4] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[5] R. M. Haralick and L. G. Shapiro. The consistent labeling problem (Part I). *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):173–184, April 1979.

[6] M. Kamada, K. Toraichi, R. Mori, K. Yamamoto, and H. Yamada. A parallel architecture for relaxation operations. *Pattern Recognition*, 21(2):175–181, 1988.

[7] H. Liu, T. Young, and A. Das. A multilevel parallel processing approach to scene labeling problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-10(4):586–590, July 1988.

[8] J. McCall, J. Tront, F. Gray, R. Haralick, , and W. McCormack. Parallel computer architectures and problem solving strategies for the consistent labeling problem. *IEEE Trans. on Computers*, c-34(11):973–980, 1985.

[9] U. Montanari. Networks of constraints: fundamental properties and applications to picture processing. *Information Sci.*, 7:95–132, 1974.

[10] B. Nudel. Consistent-labeling problems and their algorithms: expected-complexities and theory-based heuristics. *Artificial Intelligence*, pages 135–178, 1983.

[11] D. Reisis and V.K. Kumer. Parallel processing of the labeling problem. *Proc. IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pages 381–385, Nov. 1985.

[12] D.L. Waltz. Generating semantic descriptions from drawings of scenes with shadows, Rep. MAC AI-TR-271 MIT, Cambridge, MA. 1972.