A Parallel Algorithm for Colouring Graphs.

I.S. Bhandari* **C.M. Krishna****

D.P. Siewiorek*+

28 October 1987 CMU-CS-87-173

* Department of Electrical & Computer Engineering,
\+ Department of Computer Science,
Carnegie-Mellon University,
Pittsburgh, PA 15213.

** Department of Electrical & Computer Engineering,
University of Massachusetts at Amherst,
Amherst, MA 01003.
Arpa net [Bhandari] : isb@faraday.ece.cmu.edu
Phone [Bhandari] : 412-268-6638.

Abstract

An efficient parallel/distributed algorithm to do vertex colouring is presented. The algorithm is probabilistic and allows a trade-off between the number of colours used and the expected run time.

This research was supported in part by the National Science Foundation under contract CCR-8602143. The views and conclusions reported in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation, the University of Massachusetts at Amherst or Carnegie-Mellon University. We would like to thank Charles Leiserson of MIT and Edmund Clarke of CMU for helpful discussion.

1 Introduction

A very large number of resource-allocation problems can be cast into the form of the vertex-colouring problem. Examples are scheduling, register allocation for compilers, etc. Unfortunately, the problem of colouring the vertices of a graph with the minimum possible number of colours is NP-complete [1]. One has therefore to make do with suboptimal colouring algorithms. In this paper, we present a simple, parallel vertex-colouring algorithm. The algorithm is suitable for execution on parallel machines, such as the *Connection Machine*[1], or on less massively parallel computers with two or more processors. The algorithm is easy to implement on a wide variety of systems, and is versatile: it is possible, by controlling one input parameter, to trade off execution time against the number of colours used. It is useful in two ways: first, as a good graph colouring algorithm, and all that that implies; second, as a vehicle to graphically illustrate tradeoffs between communication delay and execution time in parallel algorithms.

This paper is organized as follows. Section 2 contains the model and the algorithm, and Section 3 a study of its performance in an abstract setting. Both analytical and simulation techniques are used. In Section 4, we discuss details of implementing this algorithm on the Connection Machine. The paper concludes with Section 5.

2 Model and Algorithm

2.1 Model

The basic computing element is referred to as a *Processing Element* (PE). It is any computing element which has the capacity to perform the generic operations used in describing the algorithm below. These are fairly standard operations, and we therefore do not list them explicitly.

Each vertex of the graph is associated with a PE. It will be seen that the PE's can finish the execution of the algorithm at different times. Thus, a PE can have two states: active or completed. A PE on entering the completed state does not participate further in the algorithm and is free to go its own way.

The algorithm is written for a distinguished PE. It consists of a number of logical steps which are executed in synchronism by all active PE's. In practice, this does not pose a problem, since the sending or receiving of messages in these steps can be used for synchronization.

The algorithm makes the implicit assumption that the user is prepared to tolerate $\Delta + 1$ colours, where Δ is the maximum degree over all vertices in the graph. The algorithm may end up using far fewer colours than this, but this is a matter of chance.

It is assumed that there exists a parameter, called *identity*, whose value may be used to completely order the PE's. In practice, this is usually not a problem. A parallel system will have different names or identity numbers for its individual PE's, which may be used to define an ordering on the PE's.

[1] "Connection Machine" is a trademark of Thinking Machines Incorporated.

All data structures, variables, functions, and primitives used are local. These are:

- adjacency_list – the adjacency list of the vertex represented by the PE.
- degree – the degree of the vertex represented by the PE.
- total_no_of_colours_allowed – this is never less than the degree of the vertex plus 1.
- tentative_colour(v) – this is a variable used to store the colour chosen tentatively by the PE representing vertex v.
- identity(v) – a variable used to store the identity of the PE representing the vertex v.
- gone_their_own_way – Each PE notes in this array which of its neighbours have finished colouring, and gone their own way.
- SEND – a primitive used by a PE to send messages to another PE. It is assumed that messages are sent asynchronously and buffered at the destination PE until it executes a RECEIVE.
- I_am_done – When a PE determines it has achieved an allowed colour, it sends out this signal to its neighbours. The actual value of this colour was contained in the tentative_colour broadcast of the previous loop iteration.
- set_of_allowed_colours – This is used to keep track of the set of colours which are still available.
- random[set,bias] – A function which takes an ordered set as input and returns a randomly chosen element from that set. When bias is set to zero, each element has an equal probability of being chosen. As the bias increases, the smaller-indexed elements have a greater probability of being chosen.

2.2 Algorithm for Massively Parallel Machines

This algorithm is used when the parallel machine has at least as many processors as there are nodes to colour.

Each processor maintains a set of colours it can choose from. It makes the choice randomly, and then transmits it to its neighbours (as defined by the graph being coloured). If, at any cycle, a processor has chosen a colour that has not also been chosen by a neighbour with a higher priority, this becomes the final colour of the node associated with that particular processor. If there is a clash of colours, the processor corrects the set of_allowed_colours, and tries again.

```
**this procedure is written for the PE representing vertex i **
procedure(colour);
set_of_allowed_colours:={1,2,...,total_number_of_colours};
done:=0;
gone_their_own_way:= {};
while (state = active) do
```

```
    tentative_colour(i) :=random[set_of_colours_available,bias];
    if (done=0) then
        begin
         SEND tentative_colour(i) to all neighbours believed still
         to be active;
     end
    else
        begin
         SEND "I am done" signal to all neighbours believed still
         to be active (/* The processor has completed execution,
         and is now free to go its own way*/);
        end
    end if;

    if (state = active) then
        begin
         RECEIVE tentative_colour(j) or "I am done"
         from all PE's j in the adjacency
         list such that the PE representing j is not in the
         gone_their_own_way list;
        end
    end if

    if ((tentative_colour(i) = tentative_colour(j) for all such j) OR
    (identity(i) < identity(j) for all j s.t. tentative_colour(i) =
    tentative_colour(j))) then
          done:=1;
        end
    else
        begin
           set_of_allowed_colours:=set_of_allowed_colours -
                 {k | k=colour(j) for all j from whom a "I am done"
                 signal was received in this iteration of the while
                 loop};
           gone_their_own_way:=gone_their_own_way    {k|k received
                 "I am done" signal in this iteration of
                  the while loop};
        end; else
      end; if
    end; do while
end; procedure colour.
```

2.3 Algorithm for Non-Massively Parallel Machines

The algorithm in Section 2.2 can be trivially modified for machines with fewer processors than there are nodes to colour. The only change is that there will be at least some processors which have more than one node to colour. The modification consists of having a processor colour its nodes one after the other; i.e. the processor concentrates on one node at a time, and only takes up another node when it has coloured the previous one on its list. The performance of this algorithm partially reflects, we will argue in Section 4, the cost of communication between processors.

3 Performance of the Massively Parallel Algorithm

When carrying out the analytical derivation of mean colouring time (which we express for convenience in units of the number of iterations of the `while` loop) we must keep in mind that there is a delay in information transmission. When a node i finds that none of its higher-priority neighbours has the same colour that it has chosen, it stops choosing colours. However, the neighbours of i do not have that information until the next iteration of the `while` loop, which is when i broadcasts its "I am done" signal. So, there can be one iteration of the `while` loop in which node i has been given its final colour (say c_i), but in which the lower-priority neighbours of i do not have that information, and thus do not correct their colour sets (by deleting c_i if it is in their colour set).

In some cases, the "I am done" signal from a processor i may be redundant information to another processor j: in such cases, it is possible to modify the algorithm slightly so as to make the delay in receiving the "I am done" signal less costly. The benefits of this are most apparent in densely-connected graphs. Denote by $S(i,n)$ the nodes which are of higher priority than node i and which have not sent out their "I am done" signal up to and including the n'th iteration of the `while` loop. If every node in $S(i,n)$ is a neighbour of i, the completion signal is clearly redundant as far as node i is concerned. In such cases, node i can anticipate any completion signal to be generated by a node in $S(i,n)$ and can adjust its colour set *in that iteration of the* `while` *loop*. However, this modification increases the computational burden on the processing element, and does not seem to warrant implementation.

Alternatively, the sending of the "I am done" signal and its corresponding `Receive` may be introduced as a separate step. This will mean that two messages are exchanged with every neighbour for every execution of the `while` loop, the second message being the final colour (if any) chosen. This ensures that all colour sets are adjusted in the same interation in the `while` loop. We will call this the SIMD[2] modification: it an implementation suitable for a SIMD computer such as the Connection Machine.

In the subsections that follow, we present analytical results for complete graphs, and simulation results for random graphs.

[2]SIMD = Single Instruction, Multiple Data Stream

3.1 Complete Graphs

Since every pair of nodes is connected by an edge, the complete graph has the following properties:

P1. All nodes have the same degree and thus the same colour set

P2. At every iteration of the `while` loop, the subgraph induced by the so-far-uncoloured nodes is also a complete graph, each node of which has the same colour set.

Moreover, the exact numbers assigned to each node do not matter; what determines when a node will be coloured is the *relative* numbering. These properties of the completely connected graph make it possible to find the mean colouring time relatively simply when `bias`=0, i.e. when the colour choice is truly random and unbiased.

There are two variables which describe the state of the system at any iteration: the number of nodes which have not yet been coloured, and the number of nodes which are not known by their neighbours to be coloured yet. Due to the time lag that we mentioned above, the two variables are not necessarily equal.

Denote by $T_k(n)$ the expected time to colour a completely connected graph, and by $\tilde{T}_k(n,m)$ the expected time to finish colouring a completely connected graph in which n nodes have not yet sent out their "I am done" signal, and in which m nodes have not yet reached their final colour. Clearly, $n \geq m$, and $T_k(n) = \tilde{T}_k(n,n)$. Let $\beta_\nu(n,m)$ denote the probability that in such a case, ν of the m not-yet-coloured nodes achieve their final colour in the next iteration of the `while` loop. Then, we immediately have the following recursion:

$$\tilde{T}_k(n,m) = \sum_{\nu=0}^{m} \beta_\nu(n,m)\tilde{T}_k(m,m-\nu) + 1$$

where, clearly, $\tilde{T}_k(n,0) = 1$, and $\tilde{T}_k(1,1) = 2$.

$\beta_\nu(n,m)$ can be found by using a probabilistic urn model [4]. Each colour can be regarded as an urn, and each node as a ball. A state of (n,m) corresponds to the m balls being thrown at random, each into one of n urns. Of these n urns, balls that land in a predefined set of $n-m$ "bad" urns (corresponding to the colours of nodes which have achieved colouring only in the previous iteration of the `while` loop and which have therefore not yet sent out their "I am done" signal) must be thrown again. The number of nodes which achieve colouring at this step is then the number of "good" (i.e. not "bad") urns that are occupied by a ball.

$\beta_\nu(n,m)$ is best calculated by the following recursion. Define the auxiliary function $\gamma_\nu(n,m,p)$ recursively as follows.

$$\gamma_\nu(n,m,p) = \begin{cases} \{(n-m)/n\}^p & \text{if } \nu = 0 \\ m/n & \text{if } \nu = p = 1 \\ \frac{n-m+\nu}{n}\gamma_\nu(n,m,p-1) + \frac{m+1-\nu}{n}\gamma_{\nu-1}(n,m,p-1) & \text{else if } \nu, p \geq 1 \end{cases}$$

Consider a model in which the balls are thrown in one after the other. In that case, $\gamma_\nu(n,m,p)$ can be regarded as the probability that ν out of the $n-m$ "good" urns have been occupied after p of the m balls have been thrown in. From this argument, it follows that

$$\beta_\nu(n,m) \;=\; \gamma_\nu(n,m,m).$$

Bounding the expected time for a given node to finish executing is equally simple. Clearly, the lowest-priority node has the maximum expected finishing time. Define

$$L(n,m) = \sum_{\nu=0}^{m-1} \gamma_\nu(n,m,m-1)\left\{\frac{m-\nu}{n} + \left(1-\frac{m-\nu}{n}\right) L(m,m-\nu)\right\}$$

where $L(n,0) = 1$. We leave it to the reader to show that the expected finishing time of the lowest-priority node is $L(n,n)$.

The computation of mean execution time for the SIMD algorithm is much easier. The recursion for the mean execution time is found first by ignoring the completion signal, and then adding 1 to the final result to account for it. Define the recursion

$$\hat{T}_k^M(n) \;=\; \begin{cases} 0.0 & \textit{if } n = 0 \\ 1.0 & \textit{if } n = 1 \\ \sum_{m=1}^{n} \pi(n,m)\hat{T}^M(n-m) & \textit{otherwise} \end{cases}$$

It is easy to see that the mean execution time for the SIMD algorithm is given by

$$T_k^M(n) = \hat{T}_k^M(n) + 1.0$$

Keep in mind that this is in units of the number of iterations of the `while` loop, which contains *two* transmissions in the SIMD case, while the original algorithm contains only one transmission per iteration.

The bound on the expected finishing time of the lowest-priority node for the SIMD algorithm can be computed as follows. It is easy to show that if m nodes remain uncoloured at a particular iteration, the probability that the lowest-priority node gets coloured at the next iteration is given by $\{(m-1)/m\}^{m-1}$, which decreases monotonically as $m \to \infty$, converging to $1/e$. As far as the lower-priority node is concerned, the process of colouring it is therefore a Bernoulli process with probability of success at each step at least $1/e$. It follows from this and elementary probability that the mean time to colour any particular node is upper-bounded by e, regardless of the size of the graph.

Remark 1: The perfect symmetry of the complete graph makes it possible to obtain exact results. Properties P1 and P2 make it sufficient to determine the probability that a certain number of nodes have been coloured in the first iteration of the `while` loop; it is not necessary to find the colouring probability of every set of distinguished points. Unfortunately, the complete graph is the only graph which has properties P1 and P2. For most other graphs, we have to be content with approximate solutions.

Remark 2: Despite the fact that the completion signal is always redundant, the algorithm cannot do

without it. This is because the nodes have no way of knowing that the graph is completely-connected: all each node knows is the set of nodes that it is connected to.

Remark 3: The analysis of less symmetric graphs than the completely connected is usually made impractical because of the amount of status information that needs to be stored during the numerical computation of their mean colouring time. At each step of the recursion, one would have to calculate the *joint* probability of the sets N_1, N_2, where N_1 is the set of nodes which have achieved colouring, and N_2 the set of nodes which have not yet sent out their completion signal. One would then have to calculate this for *every* one of the assignments of node priority, and average them. For graphs which are not very small, this procedure quickly becomes infeasible.

Numerical results are presented in Figure 1 for completely connected graphs. This simple algorithm is surprisingly fast: for example, it takes fewer than 11 iterations on the average to colour a 200-node complete graph when `bias`=0.

When the `bias` $\neq 0$, analytical results are impractical to obtain. The reason is that when the choice of colours (urns in our urn model) is biased, the state description of the underlying Markov chain has to contain the identites of the available and unavailable colours ("good" and "bad" urns in our urn model), not just their number. This results in a number of states that grows exponentially with the number of available colours, and makes the analysis intractable.

3.2 Random Graphs

Any conceivable exact analytical model for random graphs would require an impractically large number of states. As far as approximations are concerned, it is not possible to apply the techniques of decomposition due to the tight coupling between nodes connected together by an edge. We have therefore limited ourselves here to a simulation.

A random graph is characterized by a 2-tuple (n, p), where n is the number of nodes, and p is the probability that a pair of distinct nodes is connected by an edge.

In Figure 2, simulation results are presented for random graphs of 100 nodes. The `bias` value was set at 0.3.

The impact of the `bias` variable is shown in Figure 3. The probability that colour j is chosen by a processor is given by $exp(-\texttt{bias} \times j) / \sum_{k=1}^{c} exp(-\texttt{bias} \times k)$ where c is the number of colours available to the processor in question.

When the `bias` is increased, it is easy to see that the algorithm declines in speed because the chances of a collision are increased. However, because lower numbered colours are favoured by a high `bias` value, the number of colours used tends to decline, and approaches the number that would be used by a sequential algorithm. Indeed, as the `bias` tends to infinity, the number of colours used by this algorithm is exactly the number of colours used by the well-known sequential algorithm of Grimmett and McDiarmid [2].[3] We have therefore observed a tradeoff between the speed of the algorithm and the

[3] It was proved in [2] that the expected number of colours used in the Grimmett-McDiarmid algorithm does not exceed

number of colours used, controlled by the value of a single parameter.

4 Performance of the Non-Massively Parallel Algorithm

Unlike in the massively parallel case, it is impractical to analytically solve for the run time of even complete graphs. The reason is that any Markov-type analysis would have to include a state description which expressed the number of uncoloured nodes held by each processor at each step of the algorithm. The number of states will thus rise rapidly to an unmanageable amount. We have therefore contented ourselves here with a simulation.

Simulation results for the non-massively parallel case are provided in Figure 4. As expected, the expected run time reduces when the number of processors is increased. The marginal gain in adding a processor is greatest when the number of processors is small. There are three reasons for this. The first is that when the number of processors is small, the addition of a processor represents a relatively large increasing in the available processing power. The second reason is that when there is a small number of processors, the chances of a collision, i.e. of two processors picking the same colour for nodes which are connected by an edge, is small. For this to occur, at least two processors will have to have chosen nodes from their lists that are connected by an edge *and* have picked the same tentative colour. The third reason is that there is no communication delay within a processor, i.e. if two nodes a and b, connected by an edge, are taken up by the same processor one after the other, the final colour of a is known when b begins to be coloured: there is no one-step information transmission delay. So, when a large number of nodes are allocated to the same processor, the effect of the communication delay tends to decrease. To see this, we have Figure 5, in which we compare the performance of the algorithm with the performance it would exhibit if there was *always* a one-step delay in transmitting the fact that the final colour – even between nodes allocated to the same processor. Figure 5 gives an idea of the impact of the information transmission delay between processors. Naturally, the impact is the greatest when the number of processors is small.

5 Implementation on the Connection Machine

5.1 Some Features of the Connection Machine

We begin the discussion of the implementation with some remarks on the Connection Machine (CM). Our description is very sketchy: more details can be found in the book by Hillis [3].

The CM is a massively parallel SIMD machine. It can have up to 64K processors, each of which have local memory. The communication between processors is accomplished by using a hypercube interconnection network for message routing.

Programming the CM is done using *parallel variables*. A parallel variable names a slot in memory for every processor. Thus, if we define a parallel variable a on an $n-$processor system, each of the n

twice the minimum number of colours needed.

processors will have a slot in memory that will be referred to as a. This allows the manipulation of multiple data using a single instruction. For the sake of clarity, the version of a in processor i will be referred to as a_i.

The CM has a special class of operations that will be referred to as the SCAN operations. Using these allows one to conduct associative operations on parallel variables in an array of processors with time complexity log of length of the array. We will refer to this array of processors as the *scan array*. For example, PLUS is an associative operation. At the end of a PLUS-SCAN on a parallel variable p, over an array of k processors, and storing the results in p, $p_i = \sum_{j=1}^{i} p_j$. As an illustration consider doing a PLUS-SCAN on the parallel variable a, and storing the results in parallel variable b. Consider an array of only five processors. Assume that a is 0,1,2,3 and 4 in the processors 1 to 5 respectively. Therefore b will become 0,1,3,6 and 10 in the processors 1 to 5 respectively. COPY-SCAN is essentially a broadcast to all processors in the scan array. It can easily be implemented using the PLUS-SCAN. The value to be broadcast is placed in processor 1, while all other processors have 0's before the PLUS-SCAN.

5.2 Implementation Results

The SIMD modification of the algorithm was implemented on the CM. One difference between this modification and the algorithm in 2.2 has already been noted. Another difference that affects the result of the algorithm is that all processors chose from the same colour set in the SIMD modification. The reason is that since the machine is single instruction, executing `random[set_of_colours_available,bias]` should be the same in all processors, or else the instruction will have to be executed as many times as there are distinct colour sets. Hence the decision to have uniform colour sets. Thus the maximum degree of the graph plus one decides the cardinality of the colour set, rather than the local degree of the node. This should make the algorithm run faster.

Pseudo-random graphs were generated using a random number generator. The results of colouring these graphs is shown in Table 1. Once again, the number of times the while loop is executed is small. In fact in all the examples presented it is never executed more than five times. Note that two messages are exchanged per neighbour in every iteration. Therefore, while it appears that the run time in Figure 2b is greater, only one message is exchanged per neighbour per iteration. A fairer comparison can be made by doubling the number of iterations in Table 1a. Therefore the average run time in Table 1 may be said to be around 8, while for the algorithm in 2.2 it is around 5 for the edge probabilities that are used in Table 1a. This reflects the saving that is made possible by not waiting for the done message. Table 1 also shows the number of processors active in each iteration. It appears that the rate at which processors are done increases with the number of iterations where *rate(i)* is defined as the fraction $n(i)/n(i-1)$ for a particular row in the figure. Also, it is interesting to note that $rate(2)$ is always around a third. The second observation is not as evident. Let us assume that entries in the table are indexable by row number (and in some cases along with the iteration number). Define $nratio(j,k)$ to be the set of ratios $r(i,j,k)$ where j and k are row numbers such that $n(j,1)/n(k,1)$ is an integer and *edge probability(j)* is equal to *edge probability(k)* and $r(i,j,k) = n(j,i)/n(k,i)$ for all iteration numbers i. It appears that $r(i,j,k)$ is roughly equal to $n(j,1)/n(k,1)$ for all i. These observations might provide hints as to how an approximate analytical model may be built for the random graph case. The effect of increasing the number of colours allowed in steps of the maximum degree of the graph is shown in Table

1b. It can be seen that this does not produce any dramatic changes in the number of iterations. We see two reasons for this. Firstly, this number is quite low already, and therefore there is not much room for improvement. Secondly, since all processors are using the same colour set, the number of available colours may be far more than the degree of many nodes. For these nodes increasing the number of colours will not make much difference.

5.3 Possible Enhancements to the Algorithm

The special features of the CM may be used to speed up the algorithm. Since the number of executions of the `while` loop is small enough, we concentrate on improving the time for a single execution of the `while` loop.

The following sub-algorithms exploit the scan capability by throwing more hardware into every (logical) PE. Assume that every PE is made up of a number of processors organized as an array and one control processor.

Sending a Message: Assume that processors 1 to d of a PE's processors are dedicated to sending messages to neighbours 1 to d respectively. Thus the sending of a message to all neighbours may be accomplished in $O(1)$ time plus interconnection network routing time after the contents have been broadcast to the transmitting processors. *Copy* scan allows this to be done in $O(logd)$ time.
Receiving a Message: The content of a message is always a colour in the SIMD modification, which is run on the CM. No termination message is needed for the SIMD modification because the second message in a pair of messages is implicitly the "I am done" message. Assume that processors 1 to c receive all messages with content colour 1 to c respectively.
Detecting a Clash: Assume that the control processor chooses i as the `tentative_colour`. It then sends a message to processor i (in its group of processors) informing it about this choice. Processor i can detect a clash if it receives a `tentative_colour` from a neighbour and inform the control processor. Thus, this can be done in constant time. This assumes that multiple writes are allowed at a processor by neighbouring processors. There are ways to get around this: processor i can be replaced by d processors, each of which is dedicated to receiving messages from a particular neighbour. Then, the control processor could complete its broadcast of `tentative_colour` to these d processors in $O(logd)$ time.
Choosing a Random Colour: Since processors 1 to c receive messages about colours 1 to c, they may also be used to represent a dynamic set-of-allowed-colours set. After an iteration, some of the processors 1 to c will know that "their" colours are no longer available. There is a parallel variable sum which is intiailized to 1 in processors whose colours are still available and to 0 in other processors after final colours are received in an iteration of the `while` loop. *Aplus* scan on this parallel variable will complete a new enumeration of the remaining colours. The value of the scan in processor c will contain the total number of colours still available. A random number is now generated between 1 and their value. This random number is used to index into the new enumeration, thereby deciding the `tentative_colour` to be chosen. The algorithm is given below in pseudo-code. `self_id` is a constant such that processor i has `self_id` i. Assume that the control processor has `self_id` $c+1$.

```
pardo
result:= plus scan(processors 1 to c, sum, sum);
parif (self_id = (c+1)) then
    begin
       copy result from processor c;
       send (random[result]) to processor 1 so that it is assigned
       to newcolour;
    end;
    colourindex:= copy scan(processors 1 to c, newcolour,newcolour);
    parif ((result = colourindex) AND (sum=1)) then
       tentative_colour:=self_id;
    end if;
end if;
```

The time complexity is $O(logc)$.

6 Conclusions

We have, in this paper, presented a simple parallel algorithm for graph colouring. The key features of the algorithm are *(a)* that it operates on purely local information (i.e. each node only concerns itself with its neighbours), and *(b)* that it is conceptually very simple. In place of extensive interprocessor communication, it uses a random choice of colours, and then repeats the process in the event of two connected nodes being coloured the same.

There are many possible extensions of this work. We have conjectured that the expected time to colour complete graphs of order n (n=0,1,...) is an upper bound to the time to colour any other graph of order n. It would be interesting to prove this conjecture. Another extension would be to study the effect of providing more information about the graph to each processor. For example, each processor might spend time finding out the degree of the nodes which neighbour each of its neighbours. The result would throw some light on the value of structural information about the graph, and would be of interest from a theoretical standpoint. From a practical point of view, however, the algorithm as it stands is sufficiently fast that significant improvements in its execution time are unlikely to be possible by such means; and such extra information might best be employed in reducing the number of colours used.

References

[1] M.R. Garey and D.S. Johnson, *Computers and Intractability - A Guide to the Theory of NP-Completeness*, New York: W.H. Freeman, 1979.

[2] G.R. Grimmett and C.J.H McDiarmid, "On Colouring Random Graphs," *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 77, 1975.

[3] W.D. Hillis, *The Connection Machine,* Cambridge, MA: MIT Press, 1985.

[4] N.L. Johnson and S. Kotz, *Urn Models and Their Application*, New York: John Wiley, 1975.

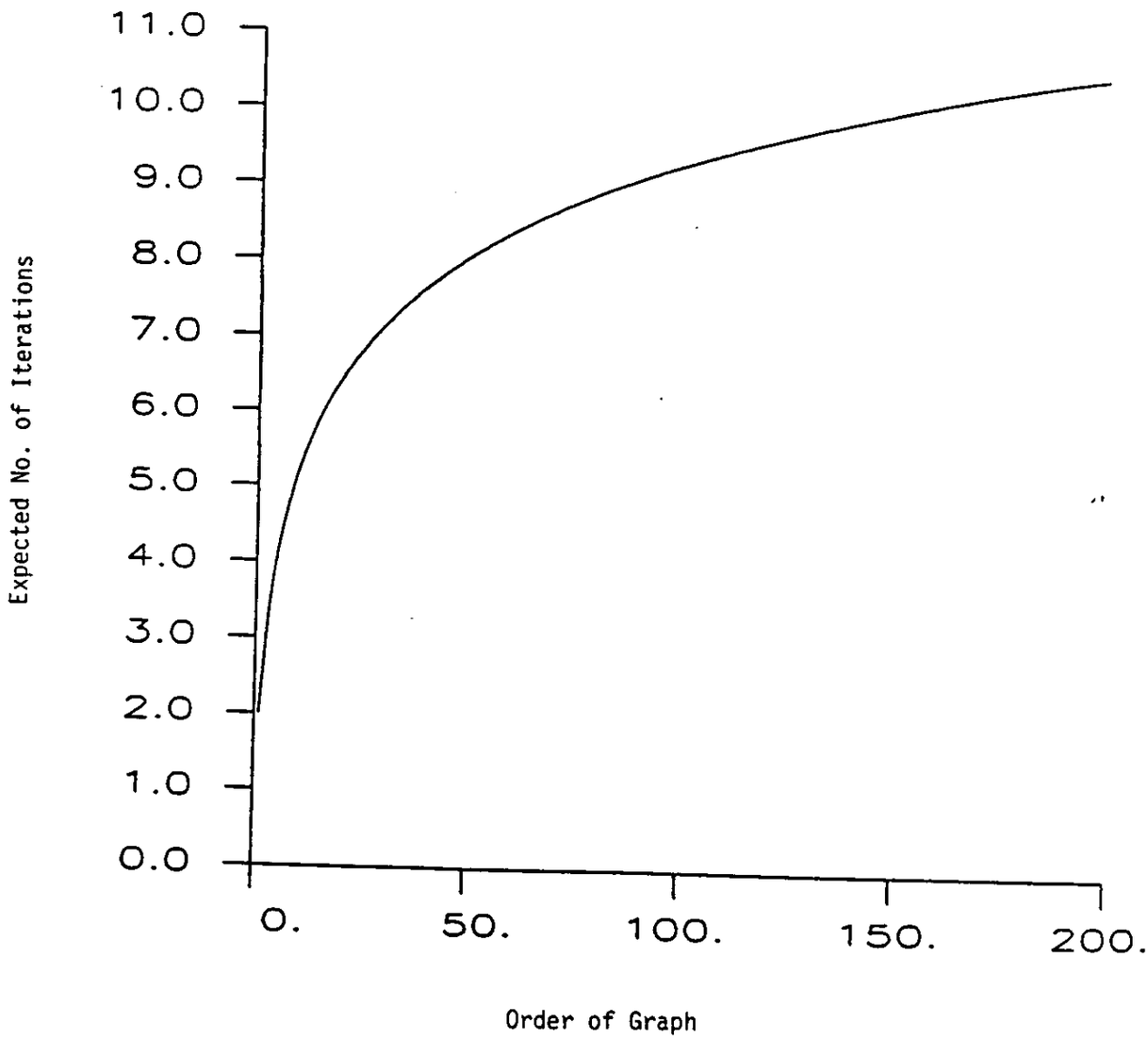

Figure 1. Expected Run Time for Competely Connected Graphs

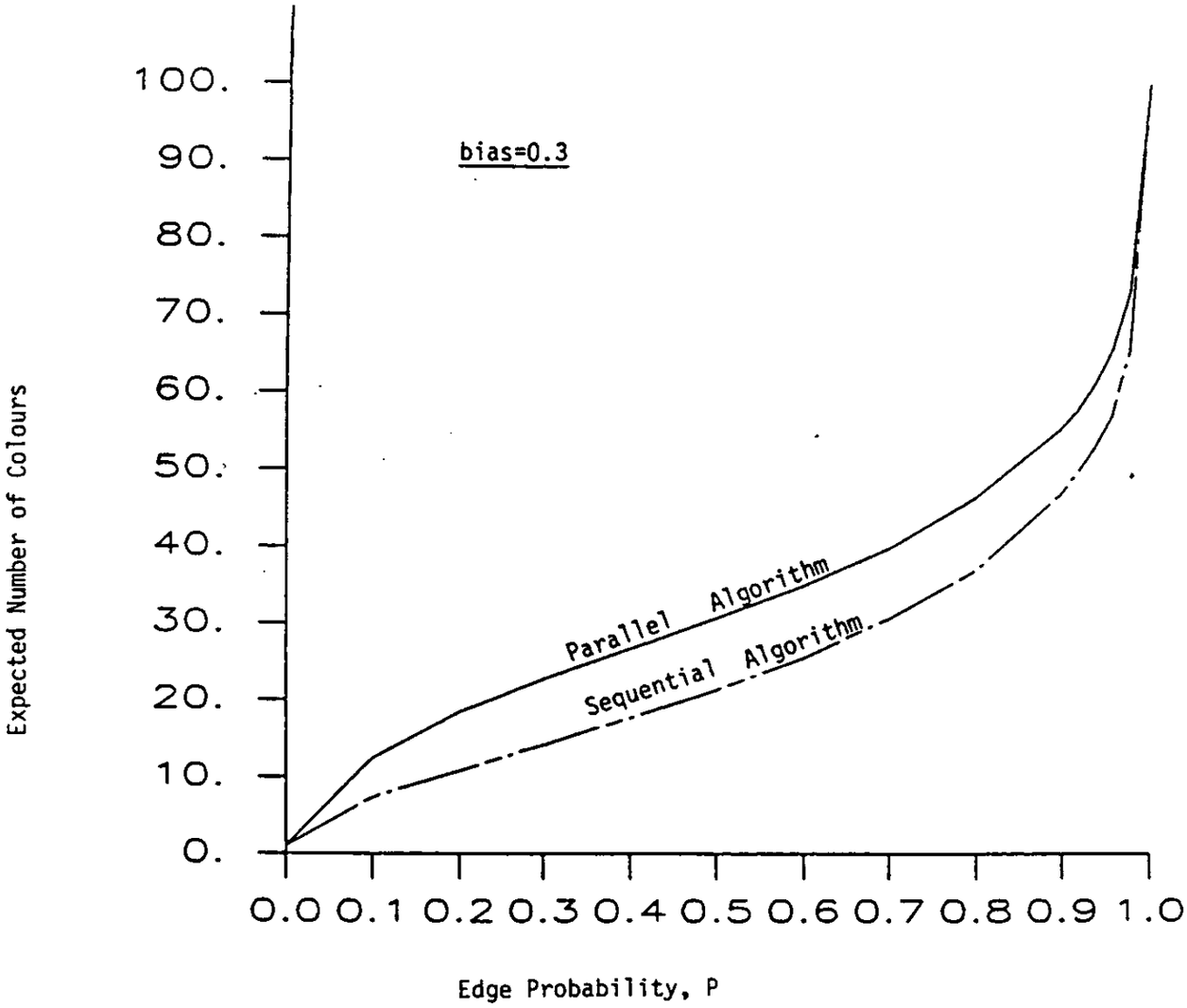

Figure 2A. Colouring a 100-Node Random Graph: No. of Colours

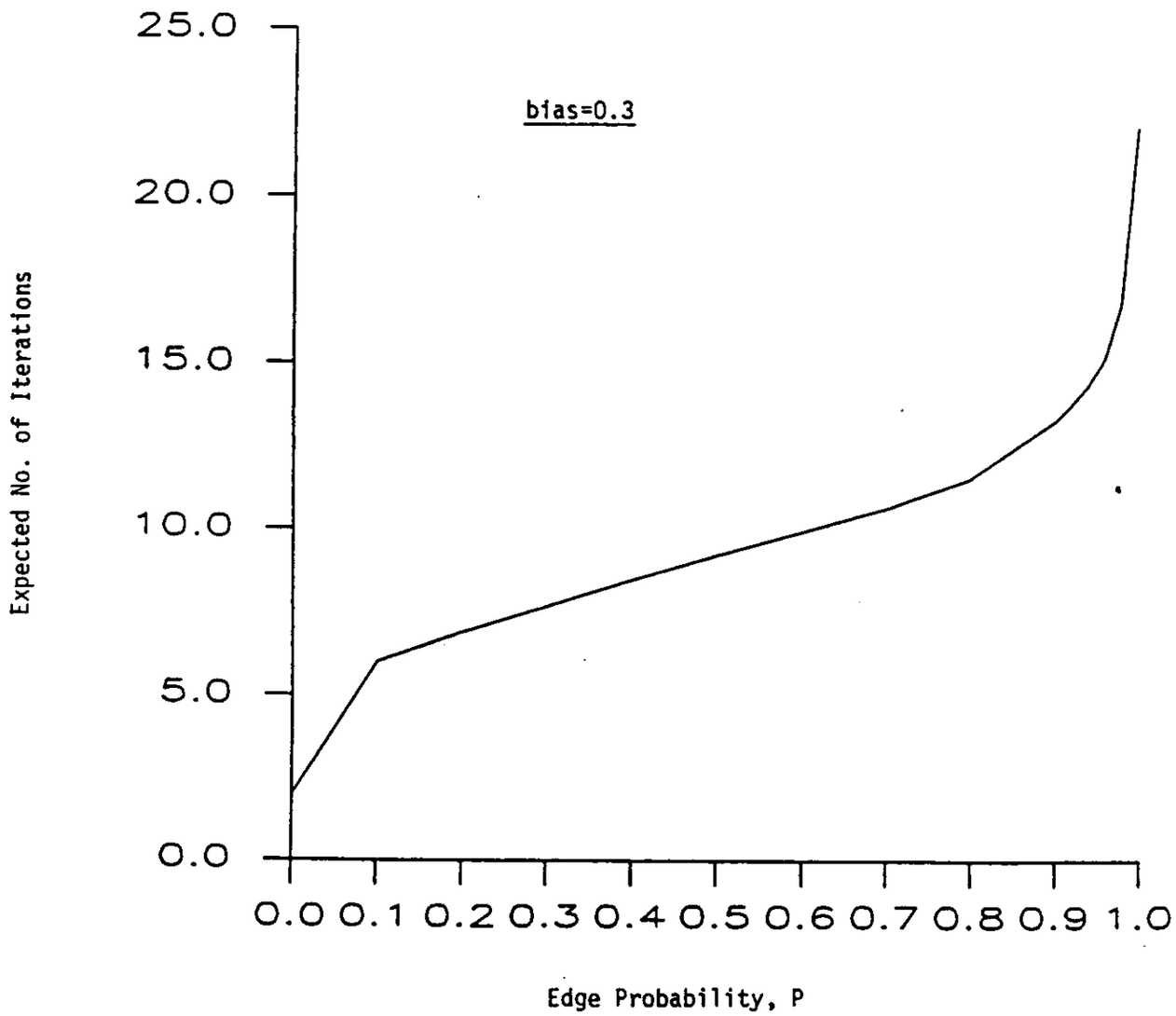

Figure 2b. Colouring a 100-Node Random Graph: Expected Run Time

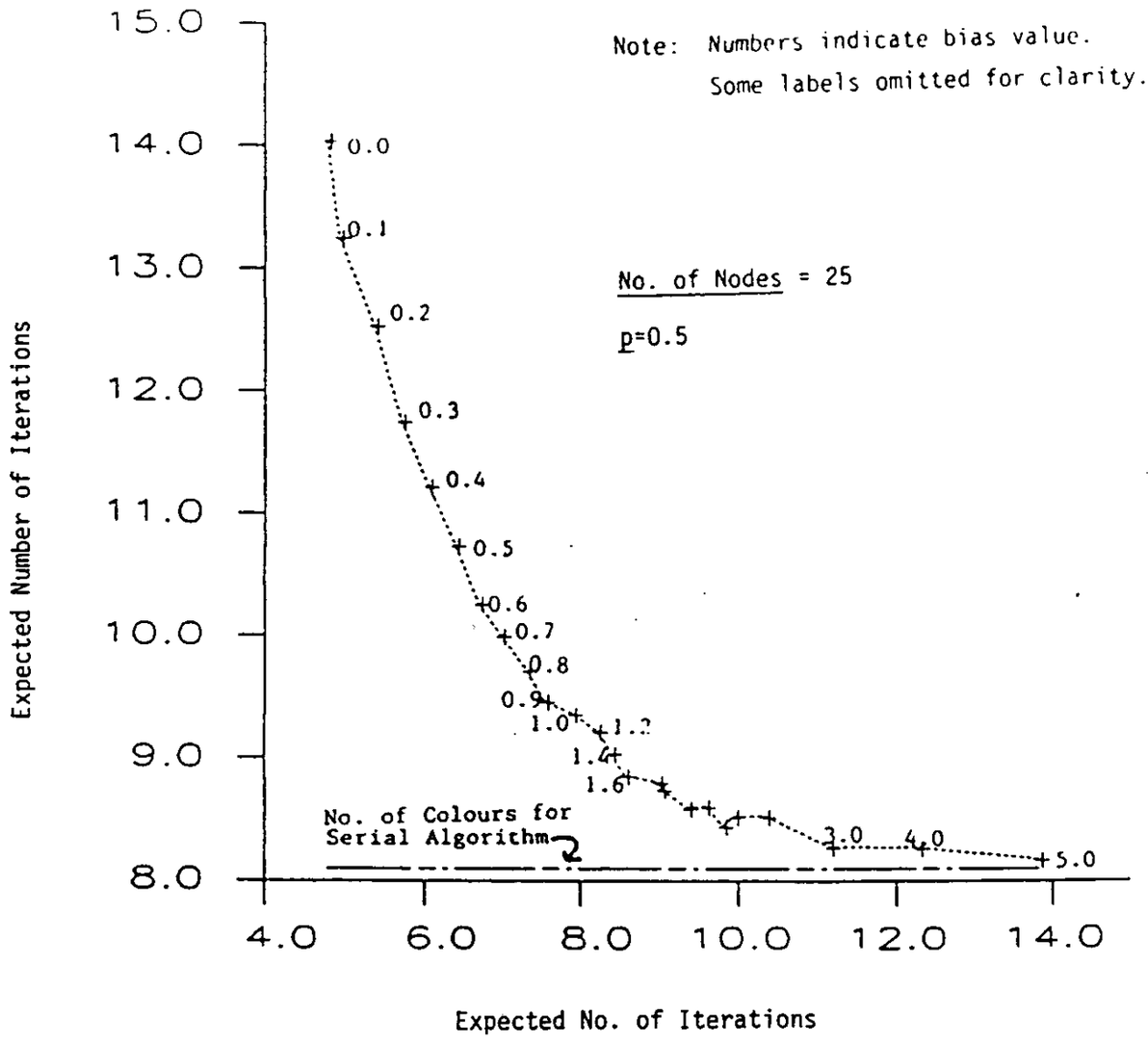

Figure 3. Controllability Using the Bias Variable

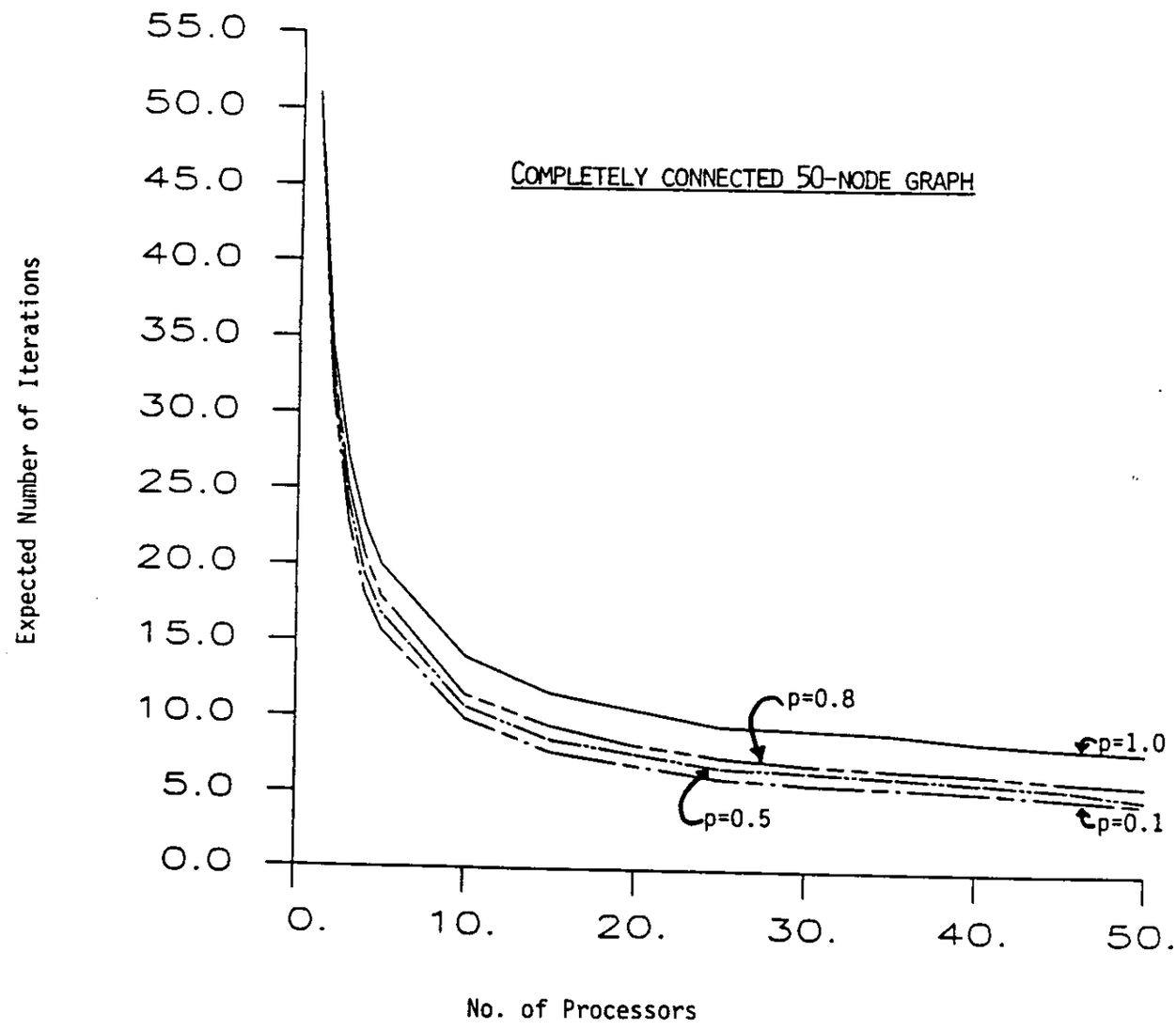

Figure 4. Performance of the Non-Massively Parallel Algorithm

n(i) Nodes Remain Before ith Execution of While Loop							
Order	edge probability*	maxdegree**	n(1)	n(2)	n(3)	n(4)	n(5)
800	0.1	107	800	275	49	4	0
800	0.02	29	800	247	41	4	0
700	0.1	95	700	235	47	6	0
700	0.02	26	700	222	29	1	0
600	0.1	81	600	197	32	4	0
600	0.02	25	600	189	25	2	0
500	0.2	129	500	197	50	8	0
500	0.1	71	500	177	41	10	2
400	0.2	103	400	123	22	1	0
400	0.1	62	400	140	27	2	0
300	0.3	108	300	118	22	1	0
300	0.2	81	300	99	11	0	0
300	0.1	43	300	93	14	1	0
200	0.5	119	200	75	12	2	0
200	0.4	100	200	74	19	3	0
200	0.3	79	200	67	11	1	0
200	0.2	52	200	58	8	0	0
200	0.1	31	200	72	10	0	0
100	0.9	97	100	34	8	1	0
100	0.8	89	100	35	7	0	0
100	0.7	75	100	35	8	2	0
100	0.6	69	100	33	8	1	0
100	0.5	59	100	38	6	0	0
100	0.4	51	100	35	9	0	0
100	0.3	45	100	30	4	0	0
100	0.2	32	100	33	5	1	0
100	0.1	19	100	36	5	1	0

* edge probability is the probability that there is an edge between any two nodes.

** maxdegree is the maximum vertex degree to be found in the graph.

Table 1a: Implementation on the Connection Machine

Effect of Increasing the Number of Colours	
Order = 800,edge probability = 0.1,maxdegree = 107	
Number of colours used	Number of Iterations
108	4
215	3
322	3
429	2
536	2
Order = 700,edge probability = 0.1,maxdegree = 95	
Number of colours used	Number of Iterations
96	4
191	3
286	2
381	2
476	2
Order = 600,edge probability = 0.1,maxdegree = 81	
Number of colours used	Number of Iterations
82	4
163	3
244	3
325	2
406	2
Order = 500,edge probability = 0.1,maxdegree = 71	
Number of colours used	Number of Iterations
72	5
143	3
214	2
285	2
356	2
Order = 400,edge probability = 0.1,maxdegree = 62	
Number of colours used	Number of Iterations
63	4
125	3
187	2
249	2
311	2

Table 1b: Implementation on the Connection Machine (cont'd)