A Comparison of Parallel Graph Coloring Algorithms

J. R. Allwright

School of Computer and Information Science, Syracuse University.
Syracuse, NY 13244, U.S.A.

R. Bordawekar, P. D. Coddington, K. Dincer

Northeast Parallel Architectures Center, Syracuse University,
Syracuse, NY 13244, U.S.A.

C. L. Martin

Department of Physics, Rice University,
P.O. Box 1892, Houston, TX 77251-1892 U.S.A.

Abstract

Dynamic irregular triangulated meshes are used in adaptive grid partial differential equation (PDE) solvers, and in simulations of random surface models of quantum gravity in physics and cell membranes in biology. Parallel algorithms for random surface simulations and adaptive grid PDE solvers require coloring of the triangulated mesh, so that neighboring vertices are not updated simultaneously. Graph coloring is also used in iterative parallel algorithms for solving large irregular sparse matrix equations. Here we introduce some parallel graph coloring algorithms based on well-known sequential heuristic algorithms, and compare them with some existing parallel algorithms. These algorithms are implemented on both SIMD and MIMD parallel architectures and tested for speed, efficiency, and quality (the average number of colors required) for coloring random triangulated meshes and graphs from sparse matrix problems.

1 Introduction

Many simulations in computational science discretize the continuum world as a triangulated mesh. This is particularly common for partial differential equation (PDE) solvers [**?**, 6]. These meshes are irregular and often adaptive, that is, they change during the course of the simulation [13]. For example, in computational fluid dynamics, the simulation of airflow over an airplane would use a finer mesh in turbulent regions and a coarser mesh in regions of laminar flow. The positions of these regions are not known initially, but must be identified during the course of the simulation.

Such dynamic, irregular triangulated meshes also occur in models of fluctuating two-dimensional surfaces, such as cell membranes and lipid bilayers [21, 9, 4]. In this context the mesh is referred to as a *dynamically triangulated random surface* (DTRS). The same type of DTRS models have recently generated a lot of interest among physicists as discrete models of string theory and quantum gravity that are amenable to numerical computation [9, 4, 2]. String theories are quantum field theories in which the fundamental particles are tiny one dimensional strings, rather than points with no dimension. In this case the two-dimensional surface modeled by the triangulated mesh is the *world-sheet* swept out by the string in some higher dimensional space-time (the *embedding* space).

Numerical calculations for these models involve integrating over all possible instances of the DTRS. This is done using a Monte Carlo technique [3, 17], which generates a new DTRS at every iteration. If we describe the mesh in terms of an undirected graph with vertices and edges, then the Monte Carlo update of the mesh is achieved using two basic operations [2]: the update (or "flip") of an edge of the graph, which changes the structure of the graph (see Figure 1); and the update of the variables associated with a vertex of the graph, which, for example, might be the position of the vertex in the embedding space, in which case the update is just a "move" of the vertex (see Figure 2). These two updates are sufficient to generate all possible triangulated meshes [2, 4].

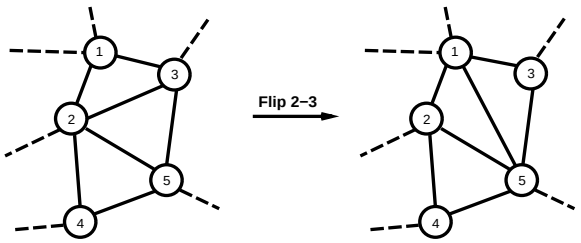

Figure 1: The operation of updating an edge, a "flip".

In DTRS simulations, both the vertex and edge update operations are generally done using the Metropolis Monte Carlo algorithm [2, 3, 17]. The update of the variables on each vertex depends only on the values at neighboring vertices.[1] The Metropolis algorithm requires that dependent (neighboring in this case) values cannot be updated simultaneously (i.e. in parallel).

[1] In general this will also depend on next-nearest neighbor vertices, but for simplicity we will consider only the nearest neighbor problem. The general case is a simple extension of this problem.

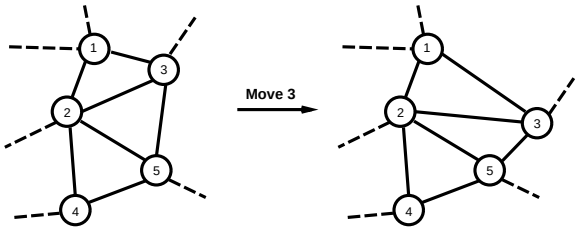

Figure 2: The operation of updating a vertex, a "move".

This is also the case for PDE solvers that use an iterative Gauss-Seidel algorithm, which has much better convergence properties than a Gauss-Jacobi algorithm for which all variables are updated simultaneously [6, 7, 14]. For a regular square grid, a parallel Gauss-Seidel algorithm can be easily implemented by using a red/black (or checkerboard) partitioning of the grid points, and updating (in parallel) all the black points, and then all the red points [14]. For the case of an irregular triangulated mesh, a similar approach requires partitioning the vertices into sets such that no set contains a pair of neighboring vertices.

This is the well-known *graph coloring* problem. An *undirected graph* G is a set of vertices V and a set of edges E. The edges are of the form (i, j) where $i, j \epsilon V$. A *coloring* of a graph G is a mapping $c : V \rightarrow \{1, 2, \ldots, s\}$ such that $c(i) \neq c(j)$ for all edges $(i, j) \epsilon E$. $c(i)$ is referred to as the *color* of vertex i. Vertices i and j are said to be *neighbors* if $(i, j) \epsilon E$. The number of vertices is denoted by V.

Note that for the DTRS simulations, we are updating both the vertices and the edges, so for a parallel implementation we need a vertex coloring and an edge coloring. In this paper we will concentrate solely on the problem of coloring the vertices of a graph.

We would like to speed up the graph coloring part of these algorithms by doing the coloring in parallel. However our goal is to reduce the run-time for the whole computation. There is a trade-off here between the time spent in coloring the graph and in updating the edges and vertices of the graph. For an adaptive grid PDE solver, many updates will generally occur between adaptive refinements of the graph which require a new graph coloring. In this case the percentage of the time spent in coloring the graph is very small, so it is worth spending more time to get a better coloring, which should improve the parallelism and reduce the update time. However for a DTRS simulation, every iteration involves an edge update, which changes the structure of the underlying graph, so the graph must be re-colored after every iteration. The graph coloring could therefore provide a substantial overhead unless it is much faster than the update time. In this case we are mainly interested in speed, and may be willing to make do with a good coloring, rather than a better coloring which takes much longer. We are therefore interested in studying and comparing a variety of parallel graph coloring algorithms.

Graph coloring is also used in many other applications, such as time-tabling and scheduling [27, 5], optimizing the calculation of sparse Jacobian matrices in PDE solvers [8], and parallel Gauss-Seidel algorithms for solving non-linear algebraic equations on irregular grids such as power networks [18]. Graph coloring can also be used to extract the greatest amount of parallelism in computing and applying a pre-conditioner for parallel iterative sparse ma-

trix algorithms such as conjugate gradient, and parallel coloring algorithms have been implemented for this purpose [20, 19]. For these problems the graph to be colored does not change, so the coloring only needs to be performed once.

For most of the applications mentioned above, finding a good graph coloring with a small number of colors is only part of the problem. We must also be concerned with load balancing, since each coloring is followed by an update step in which the variables associated with vertices of the same color are updated in parallel. Thus we need also to ensure that the number of vertices of each color on every processor is approximately the same, in order to obtain good load balance in the update step. Our goal is therefore not just to obtain a good coloring, but to obtain a *balanced* coloring, that is, to minimize the number of colors required taking into account the load balance constraint, that the number of vertices of each color be approximately the same on every processor. It may be advantageous to make do with a larger number of colors if this makes the load more evenly balanced. We will concentrate on the first of these two goals, *optimal* graph coloring and suggest how the algorithms presented may be modified to achieve *balanced* graph coloring.

2 Graph Coloring Algorithms

2.1 Previous Work

The problem of sequentially coloring an arbitrary graph has been studied extensively [5, 22, 24]. A 4-coloring is known to exist for any planar graph [1], but non-planar graphs may require a larger number of colors. Finding a coloring of a graph using the minimal number of colors is known to be an NP–Hard problem [15]. A simpler problem is to color the graph using a small number of colors, not necessarily the minimum. A number of sequential polynomial-time algorithms exist for this problem, which use relatively few colors and have good bounds on computational complexity.

Parallel algorithms have not been as extensively studied. Luby [23] gives a parallel coloring algorithm for an n vertex graph that takes average time $O(\log n)$ for the P-RAM model. Jones and Plassmann [20] improve this to give average time $O(\log n / \log\log n)$.

Here we show that some well-known sequential graph coloring algorithms, the Largest-Degree-First algorithm [27] and the Smallest-Degree-Last algorithm [24], can be readily parallelized. We compare their performance with that of Luby's Maximal Independent Set algorithm and the Jones–Plassmann algorithm. Other good sequential algorithms, notably the Saturation-Degree-Ordering [5] and Incidence-Degree-Ordering [8] algorithms, are not well-suited to parallelization, and we do not consider them here.

Many results and specialized algorithms exist for the coloring of planar graphs. The Smallest-Degree-Last algorithm is known to require no more than 6 colors for a planar graph [22]. Diks [10] describes a complicated algorithm which 6-colors a planar graph in $O(\log^2 n)$ time using $O(n^4)$ processors in the P-RAM model. Lipton and Miller [22] give a sequential algorithm which finds a 5-coloring of a planar graph in $O(n \log n)$ time.

For the DTRS simulations, we have a two-dimensional triangulated mesh, where the positions of the vertices are described in terms of an embedding space of arbitrary dimension. If the embedding space is two-dimensional, the triangulated mesh forms a planar graph

```
n = |V|
Choose a random permutation p(1),...,p(n) of numbers 1,...,n
U := V
for i = 1 to n do in sequence
    v := p(i)
    S := {colors of all colored neighbors of v}
    c(v) := smallest color not in S
    U := U - {v}
end do
```

Figure 3: The sequential greedy algorithm for coloring a graph.

composed entirely of triangular regions.[2] For higher dimensional embedding spaces, the graph will in general not be planar, for example a mesh with the topology of a torus (the common case of a two-dimensional mesh with periodic boundary conditions) is in general non-planar, although a mesh with a spherical topology does give rise to a planar graph. For sparse matrix problems, the graphs are almost always non-planar.

We therefore consider only algorithms that are completely general, that is, they will work for any (possibly non-planar) graph. We will present results for the specific case of planar randomly triangulated meshes, and non-planar graphs derived from two different sparse matrix problems.

2.2 The Sequential Greedy Algorithm

Before looking at parallel algorithms, it is helpful to consider a simple sequential algorithm, the *greedy algorithm*. This proceeds as shown in Figure 3. Here U is the set of uncolored vertices. It is not actually used in this algorithm, however we include it for reference to later algorithms.

We can give a bound on the number of colors used by this algorithm. Let us suppose that the highest-degree vertex in G has degree d. At each stage in the algorithm, the vertex to be colored can have no more than d neighbors. Hence it must be possible to color this vertex with one of the colors $1, 2, \ldots d+1$ and so the algorithm uses at most $d+1$ colors.

2.3 Parallel Graph Coloring Algorithms

Parallel algorithms for graph coloring are based on the simple observation that any independent set of vertices can be colored in parallel, where an *independent set* is a set of vertices such that no two vertices are neighbors (i.e. share a common edge). The procedure

[2] Note that this type of graph is *not* what is referred to in the graph theory literature as a *triangulated graph*, since triangulated graphs are defined to have other properties [16]. We will therefore refer to them as triangulated meshes.

```
U := V
while ( |U| > 0 ) do in parallel
    Choose an independent set I from U
    Color all vertices in I
    U := U − I
end do
```

Figure 4: Procedure for coloring a graph in parallel.

is shown in Figure 4. The differences in the algorithms boil down to how the independent set is chosen, and how its vertices are colored.

The strategy for coloring the vertices will depend on what is required from the coloring. If the aim is an *optimal* coloring, usually the smallest available color is chosen, that is, the smallest color not already being used by a neighboring vertex. If the aim is a *balanced* coloring, the least used available color might instead be chosen, to balance the number of vertices of each color.

Note that the independent set of vertices is constructed not using G, but rather the *induced* subgraph G' consisting only of the uncolored vertices U. An *induced subgraph* of G is induced by some set of vertices V', $V' \subset V$, and consists of vertex set V' and edge set $E' = \{\text{all } (i,j) \textit{ such that } (i,j) \epsilon E, \text{ and } i,j \epsilon V'\}$.

All of the algorithms presented here are local *in the graph*, in the sense that only information from neighboring vertices is required. For efficient implementation on distributed memory parallel computers, the information must also be local *in the processor grid*, so that the amount of communication is minimized. This means that the distribution of the graph over processors must be such that the number of edges crossing processor boundaries is minimized. This is the standard *graph partitioning* problem[]. This is outside the scope of this paper, and will not be addressed here.

2.4 Maximal Independent Set

The Maximal Independent Set (MIS) algorithm [23] colors the graph by repeatedly finding the largest possible independent set of vertices in the graph. All vertices in the first such set are given the same color and removed from the graph. The algorithm then finds a new MIS and gives these a second color, and continues finding and coloring maximal independent sets until all vertices have been colored. A sequential version of this algorithm finds a MIS by working through the vertices $1, 2, \ldots n$ in order and putting a vertex in the set only if it has no neighbors already in the set.

Luby has described a parallel version of the MIS algorithm [23]. The method for finding a maximal independent set is shown in Figure 5. It basically involves finding an independent set I', removing these vertices and their neighbors $N(I')$ from the graph, and iterating this procedure, accumulating the independent sets into a maximal independent set I that is obtained when all vertices are removed.

```
I := {}
V' := V
while ( |V'| > 0 ) do
    Choose an independent set I' from V'
    I := I + I'
    X := I' + N(I')
    V' := V' - X
end do
```

Figure 5: Finding a Maximal Independent Set.

Luby proposed a Monte Carlo method for constructing the independent set in parallel. All the other steps in the MIS algorithm in Figure 5 are obviously parallel and only require local information. The independent sets are found by assigning a *weight* to each vertex. The weights chosen by Luby were a random permutation of the integers $1, 2, \ldots |U|$. An independent set can be constructed in parallel by choosing all vertices whose weights are local maxima, i.e. vertices having a weight larger than any of their neighbors in the subgraph induced by U.

The parallel MIS algorithm for graph coloring follows the basic method given in Figure 4, with a maximal independent set being constructed in parallel at each step using Luby's method. The coloring is done by giving each MIS a different color.

2.5 Jones–Plassmann

Jones and Plassmann recently described a parallel coloring algorithm that improves upon the parallel MIS algorithm [20]. They pointed out that it is not necessary to create a new random permutation of the vertices every time an independent set needs to be calculated. A single set of unique random weights can be constructed at the beginning and used throughout the coloring algorithm. This can easily be done by assigning random numbers to each of the vertices and using the unique vertex number to resolve a conflict in the unlikely event of neighboring vertices getting the same random number.

The Jones-Plassmann algorithm then proceeds very much like the MIS algorithm, except that it does not find a *maximal* independent set at each step in Figure 4. It just finds an independent set in parallel using Luby's method of choosing vertices whose weights are local maxima. The other difference is that the vertices in the independent set of Figure 4 are not assigned the same new color, as they are in the MIS algorithm. Instead, the vertices are colored individually using the smallest available color, i.e. the smallest color that has not already been assigned to a neighboring vertex. This procedure is repeated using the standard method shown in Figure 4 until the entire graph is successfully colored. A description of the Jones–Plassmann algorithm is given in Figure 6.

An example of Jones–Plassmann coloring can be seen in Figure 7. The final coloring is the same as the greedy algorithm would produce if it happened to choose the vertices in order of their weights, largest-weight first.

```
U := V
while ( |U| > 0 ) do
    for all vertices v ∈ U do in parallel
        I := {v such that w(v) > w(u) ∀ neighbors u ∈ U}
        for all vertices v' ∈ I do in parallel
            S := {colors of all neighbors of v'}
            c(v') := minimum color not in S
        end do
    end do
    U := U − I
end do
```

Figure 6: The parallel Jones–Plassmann algorithm for coloring a graph.

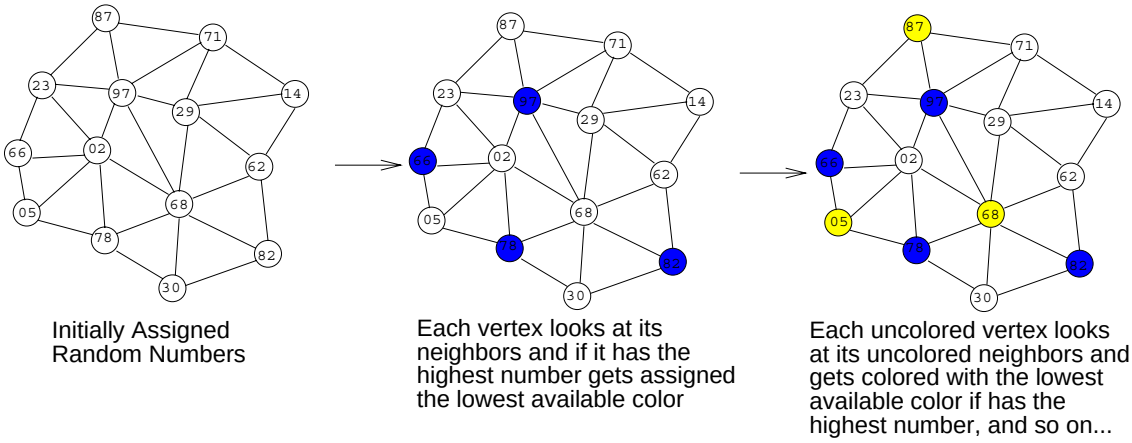

Figure 7: Coloring of the vertices during the Jones–Plassmann Algorithm.

Jones and Plassmann also gave a more detailed description of how their algorithm can work efficiently on MIMD parallel machines. They showed how the amount of synchronization required could be minimized, and also pointed out that a good sequential coloring algorithm can be used to color all the vertices which have no neighbors on different processors, with the parallel algorithm being used to color the vertices with edges that cross processor boundaries.

2.6 Largest-Degree-First

The Largest-Degree-First algorithm [27] can be parallelized using a very similar method to the Jones–Plassmann algorithm. The only difference is that instead of using random weights to create the independent sets, the weight is chosen to be the degree of the vertex in the induced subgraph. Random numbers are only used to resolve conflicts between neighboring vertices having the same degree. In this method, vertices are not colored in random order,

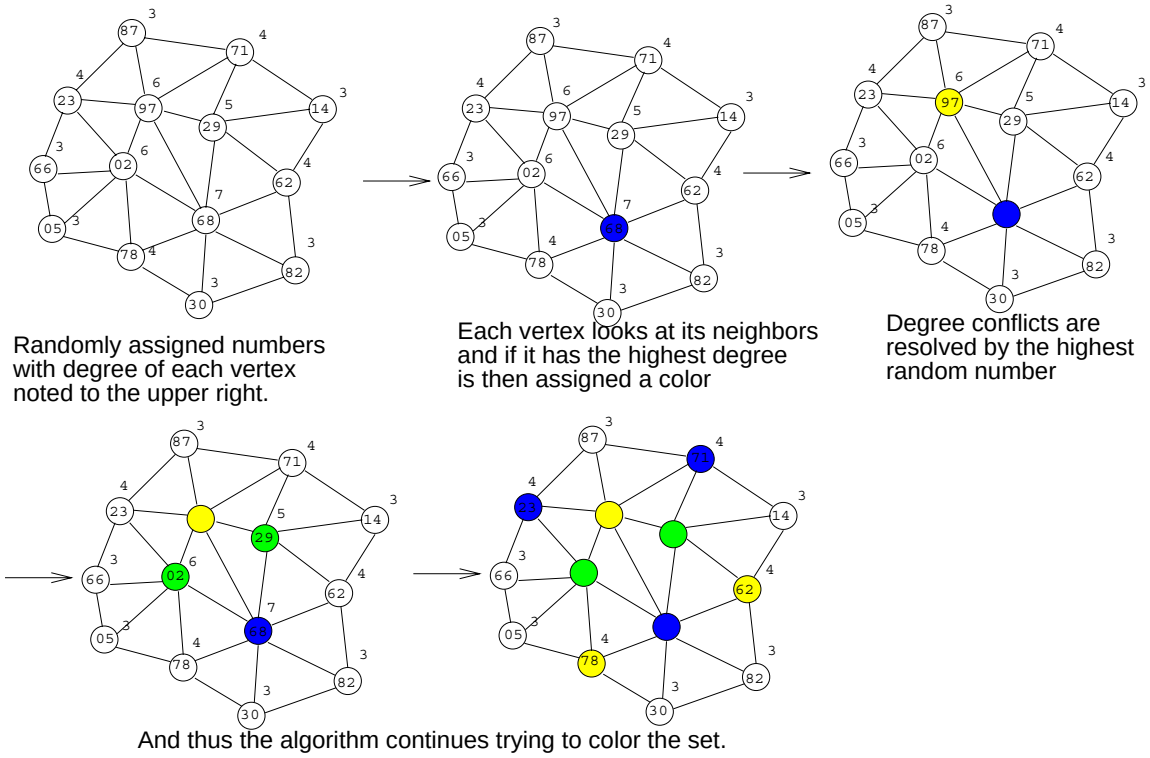

Figure 8: Coloring of the vertices during the Largest-Degree-First Algorithm.

but rather in order of decreasing degree, with those of largest degree being colored first.

This approach aims to use fewer colors than the Jones–Plassmann algorithm. A vertex with i colored neighbors will require at most color $i+1$. The Largest-Degree-First algorithm aims to keep the maximum value of i as small as possible throughout the computation, so that there is a better chance of using only a small number of colors.

An example of the Largest-Degree-First coloring algorithm can be seen in Figure 8.

2.7 Smallest-Degree-Last

The Smallest-Degree-Last algorithm [24] tries to improve upon the Largest-Degree-First algorithm by using a more sophisticated system of weights. In order to achieve this the algorithm operates in two phases, a weighting phase and a coloring phase.

The weighting phase begins by finding all vertices with degree equal to the smallest degree d presently in the graph. These are assigned the current weight and removed from the graph, thus changing the degree of their neighbors. The algorithm repeatedly removes vertices of degree d, assigning successively larger weights at each iteration. When there are no vertices of degree d left, the algorithm looks for vertices of degree $d+1$. This continues until all vertices have been assigned a weight.

This weighting process can be formalized by the parallel algorithm shown in Figure 9, which assigns a weight $w(v)$ to each vertex v. Let U be the set of unweighted vertices and $d^U(v)$ be the number of vertices in U neighboring v, i.e. the degree of the vertex in the

```
k := 1
i := 1
U := V
while ( |U| > 0 ) do
    while {∃ vertices v ∈ U with d^U(v) ≤ k} do in parallel
        S = {all vertices v with d^U(v) ≤ k}
        for all vertices v ∈ S, w(v) := i
        U = U − S
        i := i + 1
    end do
    k := k + 1
end do
```

Figure 9: The assignment of weights for the Smallest-Degree-Last algorithm.

subgraph induced by U. Clearly the degree of a vertex will decrease as its neighbors are removed from consideration. Note that the following code represents only half the coloring algorithm; once the weights have been assigned, coloring proceeds as in the Jones-Plassmann and Largest-Degree-First algorithms.

After the values of $w(v)$ are assigned, the coloring proceeds by starting at the highest value of $w(v)$ and working backwards. This coloring procedure works using the weights assigned by the first stage in the same way that the Largest-Degree-First algorithm uses the degree of the vertices. In other words the coloring phase has each vertex look around at its uncolored neighbors and when it discovers it has the highest weight (conflicts once again being resolved by a random number), it colors itself using the lowest available color in its neighborhood. A simple example of the weighting algorithm is shown in Figure 10, with the corresponding coloring shown in Figure 11.

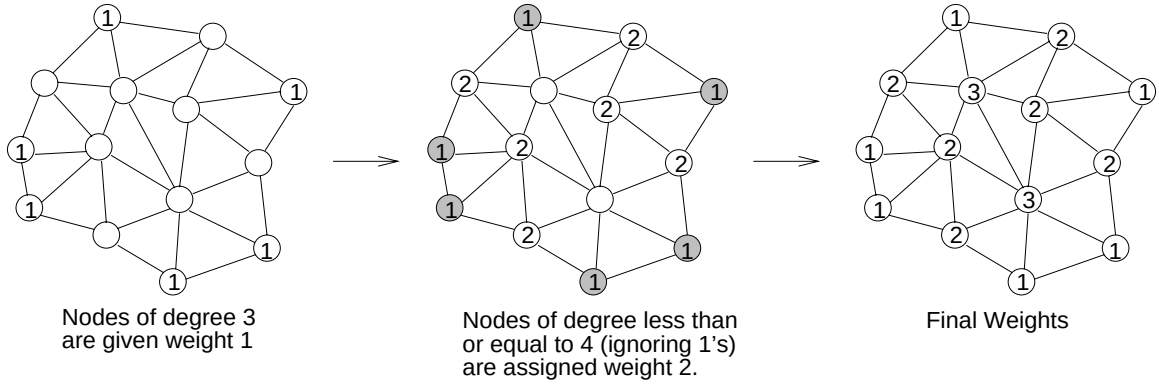

Figure 10: Weighting of the vertices during the Smallest-Degree-Last Algorithm.

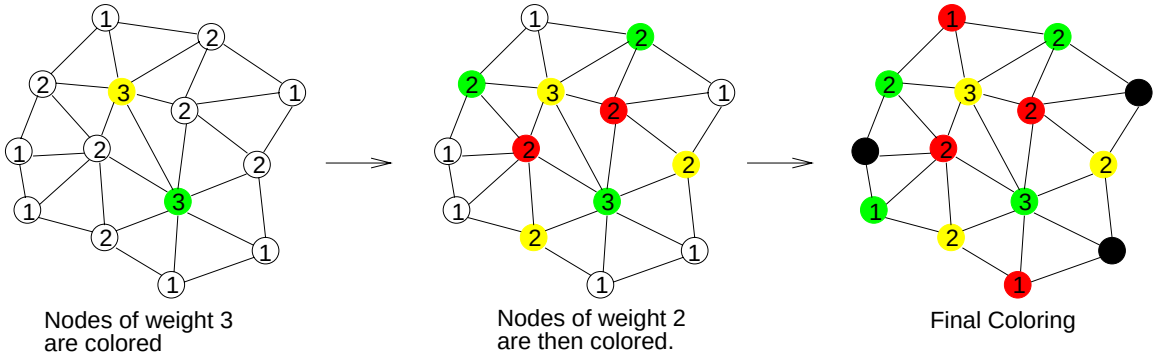

Figure 11: Coloring of the vertices during the Smallest-Degree-Last Algorithm.

3 Implementation

We are interested in finding the parallel graph coloring algorithms which give the best colorings in the least possible time. The algorithms should also be scalable, in the sense that problems of larger size (number of vertices) should be solvable just as rapidly by using a larger number of processors. Also, we hope that the number of colors required will not increase much as the number of vertices is increased.

To test the different algorithms, a set of sample random triangulated meshes of varying sizes were constructed by starting from a triangulated grid with spherical topology and flipping edges at random. These planar graphs have average degree 6. The algorithms were then run on this sample set of graphs to determine the average number of colors required to color the graphs, as well as the time (and the number of passes of the algorithm) required to arrive at the final coloring. All the results presented below are obtained by averaging over 50 sample graphs for every different problem size.

We have implemented both SIMD and MIMD versions of the algorithms. The MIMD algorithms were written using Express Fortran, a portable message passing language [25], and run on an Intel iPSC/860 computer. The SIMD algorithms were written using CM-Fortran [26], and run on a 32-node Thinking Machines CM5. A similar program was also run on a 16K node Maspar MP-1. It is interesting to see how the explicit message passing MIMD approach of the Express Fortran version compares with the SIMD virtual processor approach used with CMFortran. The MIMD algorithm does sequential coloring of vertices whose neighbors are all on-processor [20].

4 Results

The algorithms were tested on meshes with from 256 to 16384 vertices. The results on the CM5 and the MasPar MP-1 are generally similar. Figure 12 shows how the algorithms compare in terms of coloring speed. The Jones-Plassmann (J-P) and Largest-Degree-First (LDF) algorithms are significantly faster than the other two algorithms. The MIS algorithm was generally slower than these two, with the Smallest-Degree-Last (SDL) being slowest of all. Figure 13 shows how the algorithms compare in terms of the number of colors used. From the point of view of performance, the SDL was the most powerful, coloring with 5

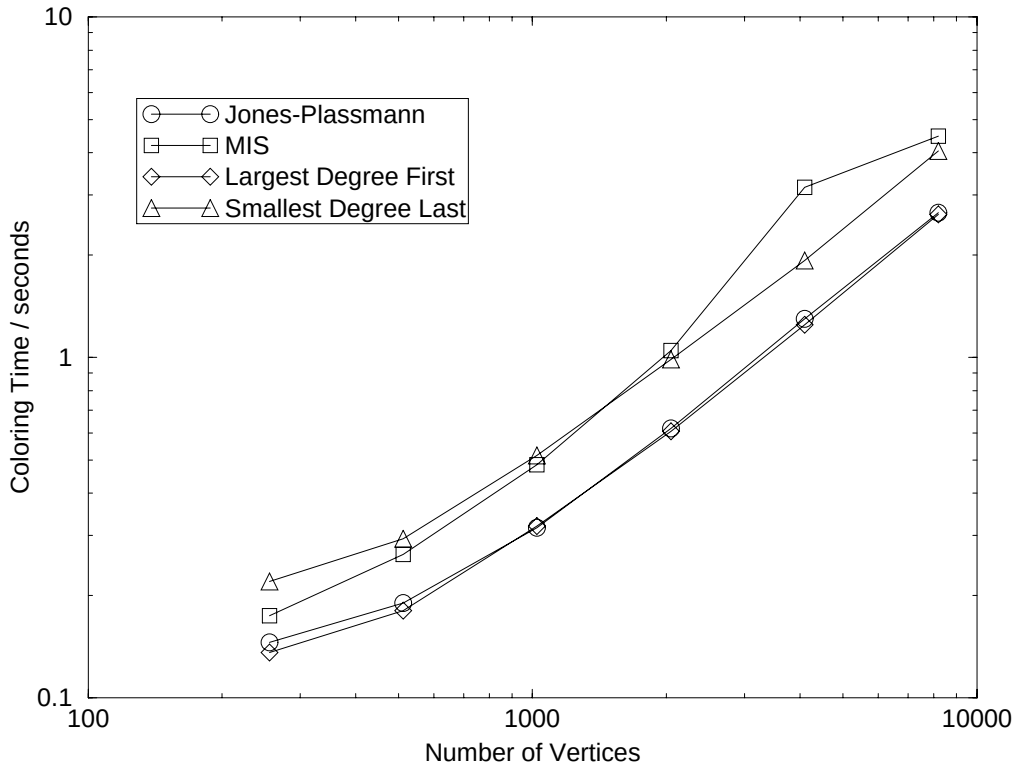

Figure 12: Processing time versus the number of vertices for the SIMD algorithm on a 32-node CM5.

colors irrespective of the graph size. The LDF algorithm was only slightly worse and the J-P and MIS algorithm were typically one color worse than LDF. The LDF, J-P and MIS algorithms all used increasing numbers of colors as the graph size increased.

Table 1 shows the results of running the various algorithms on sparse matrices from the Boeing-Harwell test set[12, 11]. The table should be compared with table 5 in Ref. [8]. The column "NP" refers to numbering given in that paper. All the colorings were performed on a 32-node CM5, with the values given being the average over 10 runs. The results on these graphs are similar to the results from the triangulated mesh problems, with SDL being the most effective algorithm, LDF being slightly less effective and J-P and MIS being roughly equal least effective. The results for ARC130, which is close to being a clique, are anomalous, with J-P and MIS algorithms performing better than LDF and SDL algorithms.

Table 2 shows the time taken by the various algorithms to compute colorings for the sparse matrices. As with the triangulated mesh problems, the SDL algorithm is clearly slowest. The LDF algorithm seems to be slightly slower than J-P and the MIS algorithm is similarly slightly slower than J-P. The results for ARC130 are again anomalous, with MIS performing significantly better than the other algorithms on this graph.

The graph in figure 14 shows the number of colors required by the algorithms running on an iPSC/860, a MIMD machine. As before, the SDL algorithm is most effective, with LDF slightly less effective and J-P least effective. Figure 15 shows how the total time for

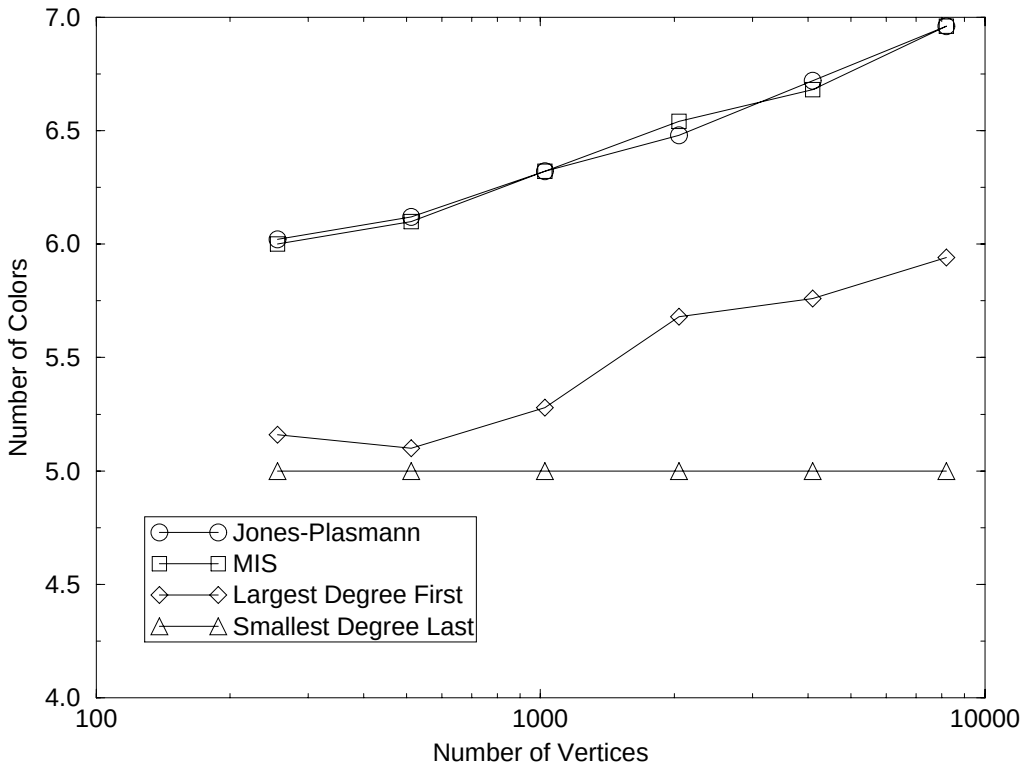

Figure 13: The number of colors required as the number of vertices is varied for the SIMD algorithm on a CM5.

the algorithms changes with the number of processors. This shows that algorithm actually slows down as the number of processors increases!

5 Conclusions

The LDF algorithm appears to perform exceedingly well in both architectures. The processing time required remains lower compared to many of the other algorithms even as the size of the problem grows larger. In particular the SDL and MIS algorithms require much more communication in each pass, which would account for their poor performance at large problem sizes and with more parallel processors. Even noting that the communication required by the LDF algorithm is equivalent to that required by the J–P algorithm, it consistently performs better. The LDF algorithm consistently achieves between 5 and 6 colors on planar graphs, which is excellent considering the NP–Hard nature of the optimal 4-coloring. The LDF algorithm is fairly easy to implement and to understand, which makes it relatively easy to incorporate into many parallel architecture codes that may be written at some point in the future.

For some algorithms, it may be important that there is no bias towards updating certain vertices before others. This is known as the requirement for *detailed balance* [3, 17]. For example, the LDF and SDL algorithms will both tend to color large-degree vertices first,

Problem	NP	Order	J-P	MIS	LDF	SDL
LUNDA	1	147	28.9	29.4	25.0	23.7
LUNDB	2	147	29.7	30.2	25.0	24.1
GENT113	4	113	20.5	20.3	20.0	20.0
IBM32	5	32	9.3	9.0	8.0	8.0
CURTIS54	6	54	12.2	12.3	12.0	12.0
WILL57	7	57	11.0	11.0	11.0	11.0
WILL199	8	199	8.4	8.3	8.0	7.0
ARC130	11	130	124.0	124.0	125.0	125.0
ASH85	28	85	12.4	11.9	10.7	10.0
ASH292	29	292	16.4	16.9	15.0	16.3

Table 1: Average number of colors required for graphs from the Harwell sparse matrix test collection.

Problem	NP	Order	J-P	MIS	LDF	SDL
LUNDA	1	147	3.0	2.5	4.2	5.2
LUNDB	2	147	3.1	2.5	4.1	5.1
GENT113	4	113	1.3	1.2	1.1	2.7
IBM32	5	32	0.17	0.19	0.18	0.27
CURTIS54	6	54	0.30	0.38	0.32	0.81
WILL57	7	57	0.17	0.23	0.20	0.35
WILL199	8	199	0.21	0.28	0.26	0.48
ARC130	11	130	47	18	47	53
ASH85	28	85	0.31	0.34	0.42	0.71
ASH292	29	292	1.1	1.1	1.9	5.9

Table 2: Time taken in seconds for coloring graphs from the Harwell sparse matrix test collection on a 32-node CM5.

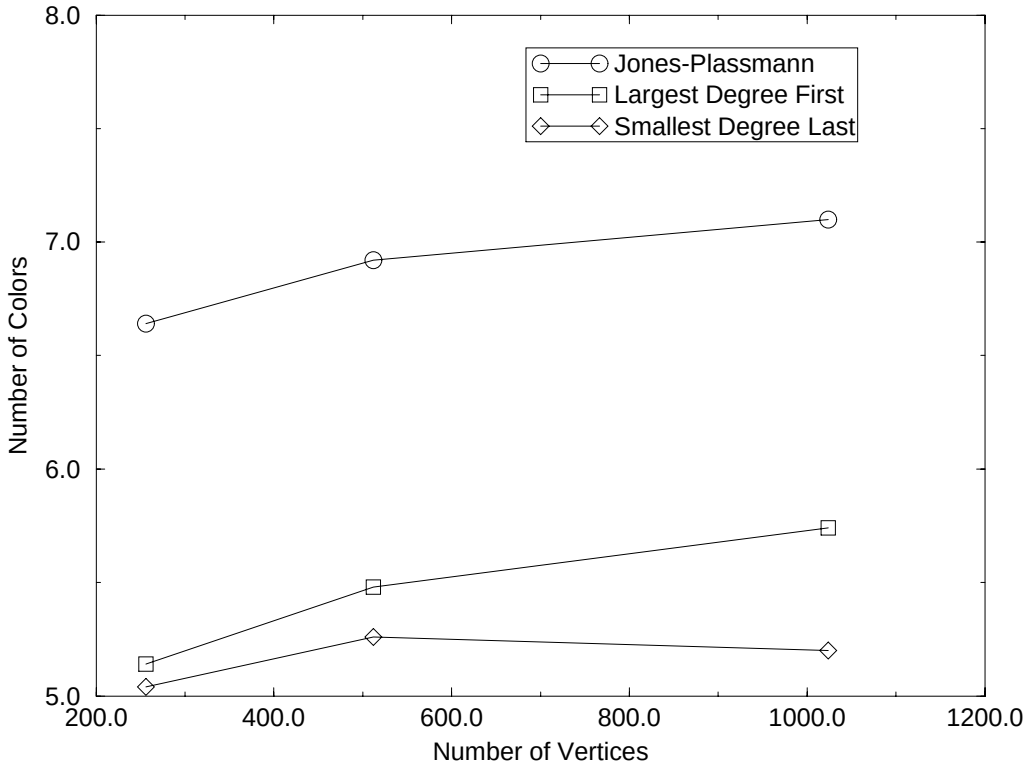

Figure 14: The number of colors required as the number of vertices is varied on an iPSC/860 using 16 nodes.

assigning them as a small number "color". If the vertices are updated in color order, then there will be a bias towards updating these vertices first. This problem can be avoided by picking the color sets to be updated in random order. With the Jones-Plassmann and MIS algorithms, this problem does not arise because there is no bias towards a particular set of vertices.

For applications such as DTRS, which require regular re-coloring of the graph, the LDF algorithm is probably the best to use, since it takes the same time as J-P, but requires fewer colors. For applications such as PDEs, the SDL algorithm, which takes longer but uses fewer colors, may well be preferable since the coloring is performed once only.

A further refinement of the coloring algorithm is a balanced coloring, requiring that the roughly equal numbers of vertices have each color. Having a small number of vertices of any one color means that there is not much parallelism to be exploited in the update step for the parallel application (such as a PDE solver or random surface simulation) that is using the results of the graph coloring. A balanced distribution of colors makes it easier to load balance the work of updating and effectively exploit a parallel machine. A simple modification to the sequential versions of the algorithms described here will achieve a balance of colors. Instead of picking the smallest available color, one picks the least used of all available colors [18]. This idea can be adapted readily to a MIMD implementation; each processor chooses from the least used color in its local patch of vertices. In a SIMD implementation this is not

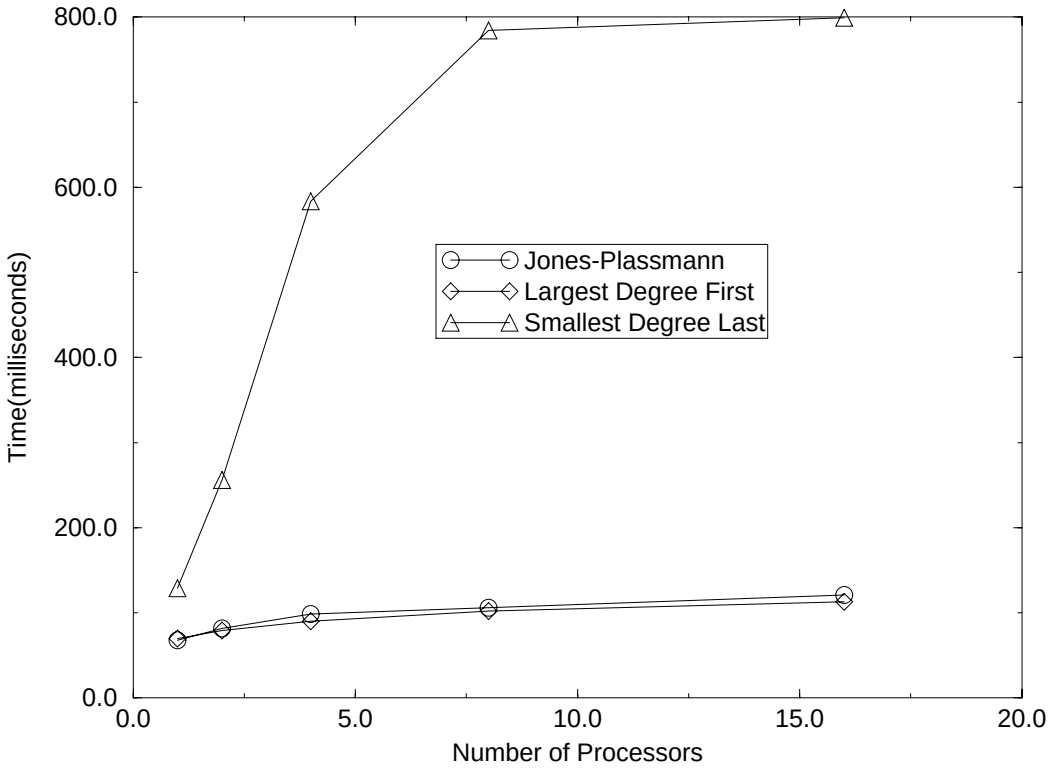

Figure 15: The coloring time for 1024 vertices as the number of processors is varied on an iPSC/860.

possible because each vertex knows only the colors of its neighbors. An alternative strategy in this case would be to pick a color at random from the set of legal colors for a vertex. This could result in a slight increase in the average number of colors required, however this cost should be outweighed by the improved load balance. No experiments with color-balancing were performed.

It was disappointing to find no speedup with the number of processors used on the iPSC/860. One reason for this may be that the algorithms as coded sent the set of weights for the whole graph to every processor and the set of colors for the whole graph to every processor. We would like to have tried larger problem sizes, which might have produced better results, but were prevented from doing so by memory limitations. It is possible that the distributed algorithm running in a larger distributed code may still be more efficient than sending the data for the graph to one processor once the communication cost of doing that is taken into account. We note that Jones and Plassmann [20] do not show a speedup, but instead show that "the running time of the heuristic is only a slowly increasing function of the number of processors used".

WE HAVE NOT DONE A GRAPH PARTITIONING!

Acknowledgments

We would like to thank Paul Plassmann, Leping Han, Sanjay Ranka, Enzo Marinari and Nikos Chrisochoides for their input to this work. Work supported in part by the Center for Research on Parallel Computation through NSF co-operative agreement No. CCR-9120008, and ARPA under contract No. DABT63-91-K-0005. Chris Martin was supported by the Research Experiences for Undergraduates Program in High-Performance Computing conducted by the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded primarily by the National Science Foundation through NSF Grant CDA-9200577, with additional funding provided by the GE Foundation Faculty for the Future Program, NPAC, and Syracuse University.

References

[1] K. Appel and W. Haken, Every planar map is four colourable, *Illinois Journal of Mathematics* **21** (1977) 429.

[2] C. F. Baille, D. A. Johnston, and R. D. Williams, Computational Aspects of Simulating Dynamically Triangulated Random Surfaces, *Computer Physics Communications* **58** (1990) 105.

[3] K. Binder and D. Heermann, *Monte Carlo Simulation in Statistical Physics*, Springer-Verlag, Berlin, 1988.

[4] M. J. Bowick and E. Marinari, Quantum Gravity, Random Geometry and Critical Phenomena, *Gen. Rel. Grav.* **24** (1992) 1209.

[5] D. Brélaz, New Methods to Color the Vertices of a Graph, *Communications of the ACM* **22** (1979) 251.

[6] N. Chrisochoides, E. Houstis, and J. Rice, Mapping Algorithms and Software Environment for Data Parallel PDE Iterative Solvers, *Journal of Parallel and Distributed Computing* **21** (1994).

[7] N. Chrisochoides, N. Mansour, and G. Fox, Performance Evaluation of Data Mapping Algorithms for Parallel Single-Phase iterative PDE solvers, To appear in Scalable High Performance Computing Conference '94.

[8] T. F. Coleman and J. J. Moré, Estimation of Sparse Jacobian Matrices and Graph Coloring Problems, *SIAM Journal of Numerical Analysis* **20** (1983) 187.

[9] F. David, A model of Random Surfaces with non-trivial critical behavior, *Nucl. Phys. B* **257** (1985) 45.

[10] K. Diks, A Fast Parallel Algorithm for Six-colouring of Planar Graphs, in *Proceedings of the 12th Symposium on Mathematical Foundations of Computer Science*, 1986, also Lecture Notes in Computer Science 233 (editors J. Grunska, B. Rovan and J. Wiedermann) p. 273 Springer-Verlag.

[11] I. S. Duff, R. G. Grimes, and J. G. Lewis, User's Guide for the Harwell-Boeing Sparse Matrix Collection, Technical Report TR/PA/92/86, Boeing Computer Services, 1992, The User's Guide and the matrices are available by anonymous ftp at orion.cerfacs.fr.

[12] I. S. Duff and J. K. Reid, Performance evaluation for Sparse Matrix Problems, in L. D. Fosdick, editor, *Performance evaluation of numerical software*, pages 121–135, North Holland, Amsterdam, 1979.

[13] G. Erlebacher and P. R. Eiseman, Adaptive Triangular Mesh Generation, *AIAA Journal* **25** (1987) 1356.

[14] G. Fox *et al.*, *Solving Problems on Concurrent Processors, Vol. 1*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[15] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman, New York, 1979.

[16] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980.

[17] H. Gould and J. Tobochnik, *An Introduction to Computer Simulation Methods, Vol. 2*, Addison-Wesley, Reading, Mass., 1988.

[18] G. Huang and W. Ongsakul, An Efficient Task Allocation Algorithm and Its Use to Parallelize Irregular Gauss-Deidel Type Algorithms, in *8th International Parallel Processing Symposium*, Cancun, Mexico, 1994.

[19] M. T. Jones and P. E. Plassmann, Scalable Iterative Solution of Sparse Linear Systems, Preprint MCS-P277-1191, 1991.

[20] M. T. Jones and P. E. Plassmann, A Parallel Graph Coloring Heuristic, *SIAM Journal of Scientific Computing* **14** (1993) 654.

[21] R. Lipowski, The conformation of membranes, *Nature* **349** (1991) 475.

[22] R. J. Lipton and R. E. Miller, A Batching Method for Coloring Planar Graphs, *Information Processing Letters* **7** (1978) 185.

[23] M. Luby, A simple parallel algorithm for the maximal independent set problem, *SIAM Journal on Computing* 4 (1986) 1036.

[24] D. W. Matula, G. Marble, and J. D. Isaacson, *Graph Coloring Algorithms*, Academic Press, New York, 1972.

[25] Parasoft Corporation, 2500 E. Foothill Blvd., Pasadena, CA 91107, *Express Fortran User's Guide and Reference Manual*, 1987.

[26] Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1264, *CM Fortran User's Guide and Reference Manual*, 1990.

[27] D. J. A. Welsh and M. B. Powell, An Upper Bound for the Chromatic Number of a Graph and its Application to Timetabling Problems, *Computing Journal* **10** (1967) 85.