Architectures, Languages and Patterns
P.H. Welch and A.W.P. Bakkers (Eds.)
IOS Press, 1998

Parallel Graph Coloring using JAVA

Thomas UMLAND*
Deutsche Telekom Berkom GmbH, Goslarer Ufer 35,
10589 Berlin, Germany

Abstract. In this paper a parallel, pipeline oriented version of a well-known sequential graph coloring heuristic is introduced. Runtime and speedup results of an implementation in JAVA on a four processor machine are presented and discussed.

1 Introduction

The idea of this paper is to introduce a parallel version of a well-known sequential graph coloring heuristic which could be easily implemented on a shared memory multiprocessor system when using an appropriate programming language like for instance JAVA [1]. The next section shall help to understand the sequential algorithm. In section 3 the parallel version is presented while in section 4 runtime and speedup results of a JAVA implementation running on a four processor SUN SPARC Workstation are given.

2 The sequential first fit algorithm

For a graph $G = (V, E)$ with vertices $V = \{v_1, \ldots, v_n\}$ and edges $E \subseteq V \times V$ a function $f : V \to \{1, \ldots, k\}$, $v \in V \mapsto f(v)$ is called a *coloring* (of the vertices) of G if for all pairs of vertices $u, v \in V$, $u \neq v$, $(u, v) \in E \;\Rightarrow\; f(u) \neq f(v)$. If we call the value $f(v)$ the *color* of the vertex $v \in V$ then coloring the graph $G = (V, E)$ simply means that every vertex of G has to be assigned a color with the restriction that adjacent vertices – i. e. those connected by an edge – must get different colors. The minimal value of k so that $f : V \to \{1, \ldots, k\}$ is a coloring of the graph $G = (V, E)$ is called the *chromatic number* of that graph.

Coloring the vertices of a graph is a problem needed to be solved in a variety of applications e. g. scheduling, register allocation, printed circuit testing etc. Unfortunately, the task of finding a coloring with the minimal number of colors is not solvable in polynomial time for an arbitrary graph (as known so far). Therefore one is looking for coloring algorithms, so-called heuristics, which do not always find an optimal coloring but have polynomial runtime.

The *first fit* algorithm is a well-known heuristic with polynomial runtime for coloring the vertices of a graph. It requires an initial ordering of the vertices of the input graph. The first vertex according to this order gets the color 1 while the other vertices are processed sequentially, assigning each vertex the least possible color which does not produce a conflict with the vertices already colored. For a detailed description see for example [4, 5].

*Author's new address since 1 March 1998 is: Deutsche Telekom AG, Entwicklungszentrum Nord, Willy-Brandt-Platz 3, 28215 Bremen, Germany, E-Mail: Thomas.Umland@telekom.de.

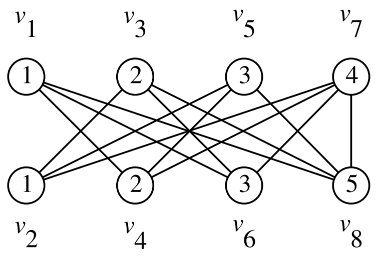

Figure 1: First fit coloring of a graph with 8 vertices.

Figure 1 shows a graph with 8 vertices for which the algorithm produces a coloring with 5 different colors. In this figure each circle symbolizes a vertex and each line an edge between two vertices. The name of each vertex is written outside and the color produced by the algorithm is written inside the circle. From the indices of the vertices the initial ordering can be deduced, i. e. vertex v_i is the i-th vertex of the ordering.[1] Because the algorithm is not too complicated this example shall suffice for illustrating its operation.

3 Parallelizing sequential first fit

Although the first fit algorithm is often called inherently sequential because of its strictly ordered procedure, we will now present a parallel variant which in practice yields not too bad speedup results on a parallel machine.

3.1 Basics

To get an idea how our parallel algorithm works we take at first a look at a possible implementation of the sequential first fit algorithm. The coloring of a vertex v_i could be implemented using two main steps:

1. Determine a list of all possible colors for v_i i. e. exclude those colors already used by vertices v_j, $j < i$ adjacent to v_i. This could be implemented using a boolean array L_i – called the possibility list of vertex v_i – with the property $L_i[k] = \text{FALSE} \Leftrightarrow \exists v_j$ with $j < i$, $(v_i, v_j) \in E$ and $f(v_j) = k$.

   This step can be performed by a procedure Build(L_i, v_j) which excludes the color of vertex v_j from the possibility list L_i of vertex v_i,

2. Determine the smallest of all possible colors for vertex v_i, i. e. look for the smallest entry in L_i with $L_i[k] = \text{TRUE}$ and assign color k to v_i.

   We put this step into a procedure Color(L_i, v_i) which colors vertex v_i in dependence on its possibility list L_i.

To color a graph with n vertices we therefore have to execute the following actions:

[1]Obviously the given graph could be colored using only 2 colors while first fit produces a coloring with 5 colors – as mentioned earlier the first fit heuristic does not always find the optimal coloring. Among others in [4, 5, 6] bounds for the difference of the chromatic number of a graph and the number of colors produced by first fit can be found. But this topic is not of interest in this context.

- Color(L_i, v_i) for $i = 1, \ldots, n$ and
- Build(L_i, v_j) for $i = 1, \ldots, n$ resp. $j = 1, \ldots, i-1$.

Of course these actions cannot be executed all in parallel because there are time dependencies resulting from the access to the possibility lists:

1. For all $j < i$ the action Build(L_i, v_j) must be executed before Color(L_i, v_i) and
2. Color(L_i, v_i) must be executed before Build(L_j, v_i) for $j > i$.

3.2 A first parallel approach

Next we distribute these actions over n processors and get a parallel algorithm which overall needs $2n - 1$ steps. To make the description easier we distinguish between *even* and *odd* steps:

1. For $1 \leq i \leq n$ during the odd steps $2i - 1$ the action Color(L_i, v_i) is executed on processor P_i;

   in parallel to these actions Build(L_{2i-j}, v_j) are executed on processors P_j for $1 \leq j < i$.
2. During the even steps $2i$ for $1 \leq i < n$ the actions Build(L_{2i-j+1}, v_j) will be executed on processors P_j with $1 \leq j \leq i$.

It can be easily verified that this distribution of the actions over the processors does not violate the above time dependencies. Of course, this algorithm could also be derived in a more formal way from the recurrences implied by the time dependencies. This method is demonstrated e. g. in [3] where parallel algorithms for the matrix multiplication are obtained from a set of recurrences describing the multiplication. However, because the dependencies here are quite easy to understand we omit this step. Figure 2 shows the distribution of the actions over 5 processors for a graph with 5 vertices. The actions of each column in the picture have to be executed sequentially on the corresponding processor while the actions listed in each row can take place in parallel at the specified time step of the algorithm.

The arrows in the picture indicate the points where the control over a possibility lists changes between two processors. In this representation we see that the control over the possibility lists flows through the processors in a pipelined fashion. We also see the typical pipeline behavior at the beginning and at the end of the execution where the pipeline has to be filled resp. emptied and therefore only a few processors are busy.

3.3 A generalized parallel approach

The first parallel approach of section 3.2 has the major disadvantage that it requires as many processors as there are vertices in the graph. The generalized version presented now allows any number of processors $P_1, \ldots, P_N$ ($1 \leq N \leq n$) to be used. In this case every processor is responsible for coloring a whole subgraph with n/N vertices instead of a single vertex in the previous version of the algorithm.[2]

[2]To simplify the notation we assume that N divides n.

Step	**Processor$_1$**	**Processor$_2$**	**Processor$_3$**	**Processor$_4$**	**Processor$_5$**
1.	Color(L_1, v_1)				
2.	Build(L_2, v_1)				
3.	Build(L_3, v_1)	Color(L_2, v_2)			
4.	Build(L_4, v_1)	Build(L_3, v_2)			
5.	Build(L_5, v_1)	Build(L_4, v_2)	Color(L_3, v_3)		
6.		Build(L_5, v_2)	Build(L_4, v_3)		
7.			Build(L_5, v_3)	Color(L_4, v_4)	
8.				Build(L_5, v_4)	
9.					Color(L_5, v_5)

Figure 2: Parallel first fit with 5 vertices and 5 processors.

The execution of the generalized parallel algorithm is illustrated in figure 3. One can see clearly the partition of the graph into N blocks with n/N vertices each. Each processor now has to color all the vertices of his corresponding block under consideration of the possibility lists prepared on the previous processors. This action is done by a procedure again named Color. The action Build(L_i, V_j) used in the generalized version performs the exclusion of the colors of *all* vertices $V_j = \{v_{1+(j-1)n/N}, \ldots, v_{jn/N}\}$ contained in the j-th subgraph from the possibility list L_i of vertex v_i which will be colored later by another processor.

Again the pipelined behavior of the algorithm which is caused by the flow of control over the possibility lists can be seen in the picture. Compared to the first approach the last phase of the algorithm where the pipeline empties is not as long in this version.

If you interpret the picture of figure 3 as a state/time diagram you recognize that only roughly half of the possible computing resources are used by this algorithm – due to the filling and emptying process of the pipeline many processors are idle for quite a long time. Therefore the speedup achieved by an implementation is not expected to significantly exceed half of the number of processors used. Nevertheless the resulting efficiency of about 50% is not too bad for this type of algorithm.

4 Runtime results of a JAVA implementation

In the generalized parallel first fit algorithm of section 3.3 we assumed that the algorithm runs on N *processors*. Nevertheless that approach is still valid if we take N as the number of concurrent *processes* – so-called threads in JAVA – which are scheduled by the operating system on possibly less than N physical processors. Therefore we can combine any number of concurrently running threads with any number of physical processors.

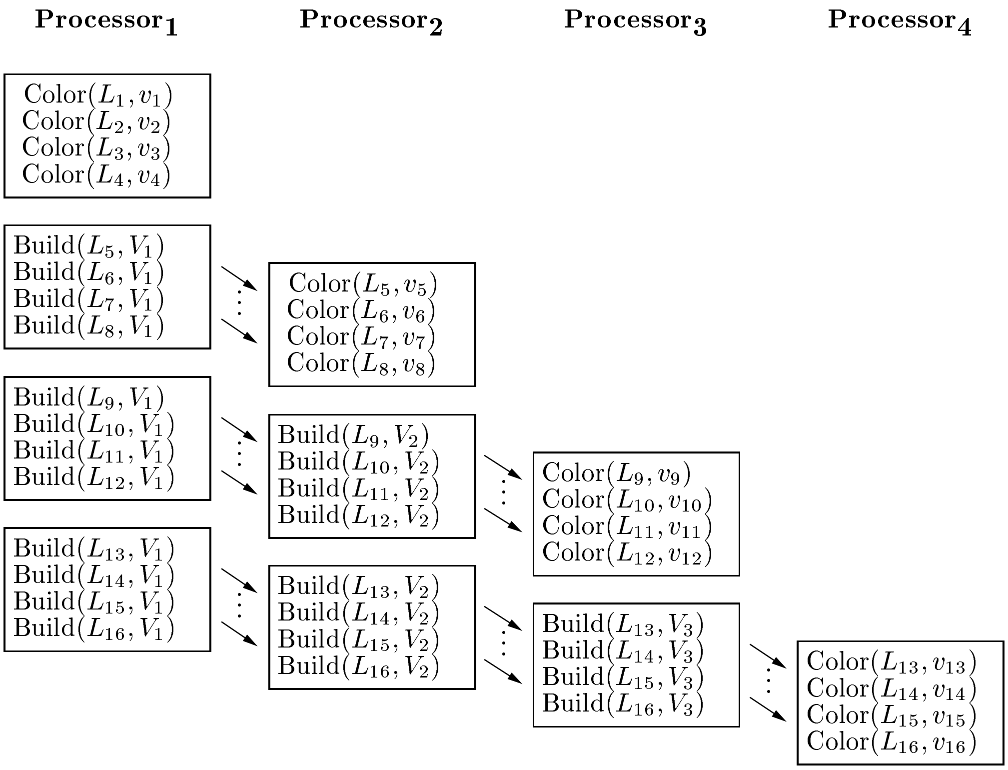

Figure 3: Generalized parallel first fit (16 vertices, 4 processors).

The representation of the graph as a boolean $n \times n$ matrix as well as the possibility lists of the vertices and the list of the already determined colors are located within an object of a new class called Graph. The actions Build and Color which have to be performed by each process can be implemented easily in JAVA and are also integrated as methods in the Graph class.

As the JAVA programming model allows shared objects between threads we generate only one (shared) instance of the Graph class which is accessible by all N concurrently running threads. The flow of control over the possibility lists – illustrated as arrows in figure 3 – is implemented by passing tokens from thread to thread via objects of a channel class CHAN which implements a directed point-to-point connection between exactly two threads. This is similar to the way communication takes place in the programming language OCCAM [2]. The class CHAN provides one method for sending and another for receiving data over the channel in order to hide the explicit synchronization constructs available in JAVA. The implementation of the CHAN class is similar to that used in JavaPP[3] which provides a lot more OCCAM and CSP mechanisms via JAVA packages. As our JAVA implementation of the coloring algorithm uses a shared graph object we do not really need to send data objects via the channels but instead transmit only a token to pass the control over a possibility list to the next thread. Therefore our channels are used simply as an elegant way to synchronized two threads in a rendezvous fashion.

The JAVA version used for this implementation was the JDK 1.1.5 with native thread support i. e. concurrently running threads are scheduled over the available processors by the operating system; a just-in-time compiler was not available. Finally the

[3]See `http://www.cs.bris.ac.uk/~alan/javapp.html` for details.

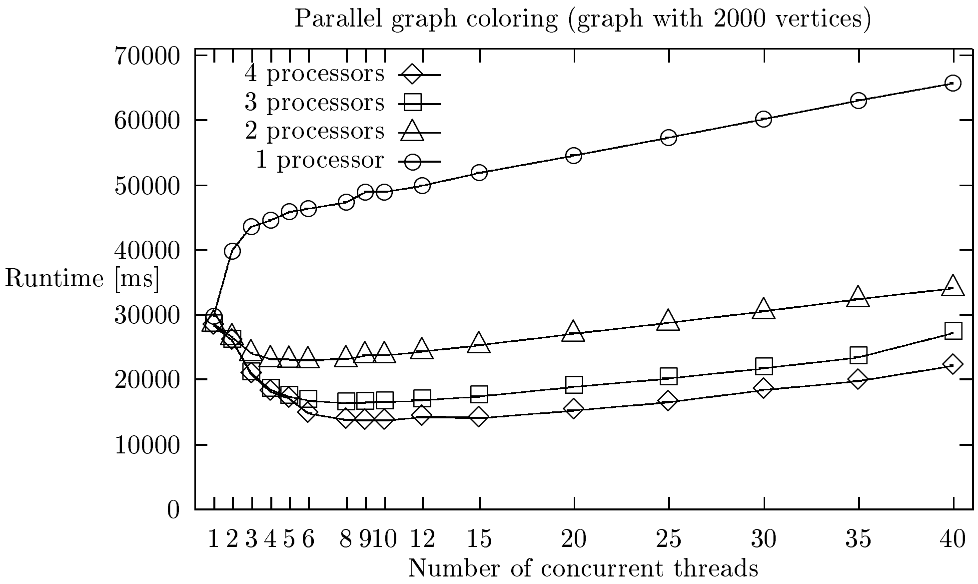

Figure 4: Runtime of the parallel algorithm measured on 1–4 processors.

implementation was tested on a multiprocessor workstation with four processors of the type SPARC-40 MHz and 128 MBytes of memory running SOLARIS 2.5.1.[4]

The kind of graph used as input has a similar structure to that shown in figure 1. It has 2000 vertices, 999001 edges and first fit yields a coloring with 1001 colors; the structure and size of the input graph has only an impact on the absolute execution time but *not* on the speedup behavior of the algorithm as experiments showed. The runtime needed to color this graph was measured for a different number of concurrently running threads N. The scheduling of the threads and the distribution over the physical processors has been carried out by the operating system. At first the program was allowed to use all four processors, then the same experiment has been repeated while disabling one, two and finally three processors. The runtime results are shown in figure 4: Each curve[5] corresponds to a fixed number of physical processors and shows the execution times for $N = 1, \ldots, 40$ concurrently running threads. The diagram shows that the program indeed runs faster when using more than one processor.

To measure the behavior of the parallelization, the speedup for each curve relative to the runtime of one thread has been calculated and is shown in figure 5. The diagram clearly shows different speedups depending on the number of processors used. If we have only one physical processor in operation and start more than one thread only semi-parallel execution is possible; therefore the time needed to administrate and switch the threads is dominant and slows down the obtainable speedup. This effect can be seen in the lower-most curve of the diagram. When using more physical processors they could be utilized to execute threads in parallel and the speedup increases. In this case the administration overhead is only dominant if the number of threads is much greater than the number of processors. As expected the speedup is better if more physical

[4]Although nowadays this is a quite slow machine, the measured speedup effects should be independent of the processors speed and are expected to be the same on machines with more up-to-date processors.

[5]The separate data points have only been connected to ease the identification of points belonging together.

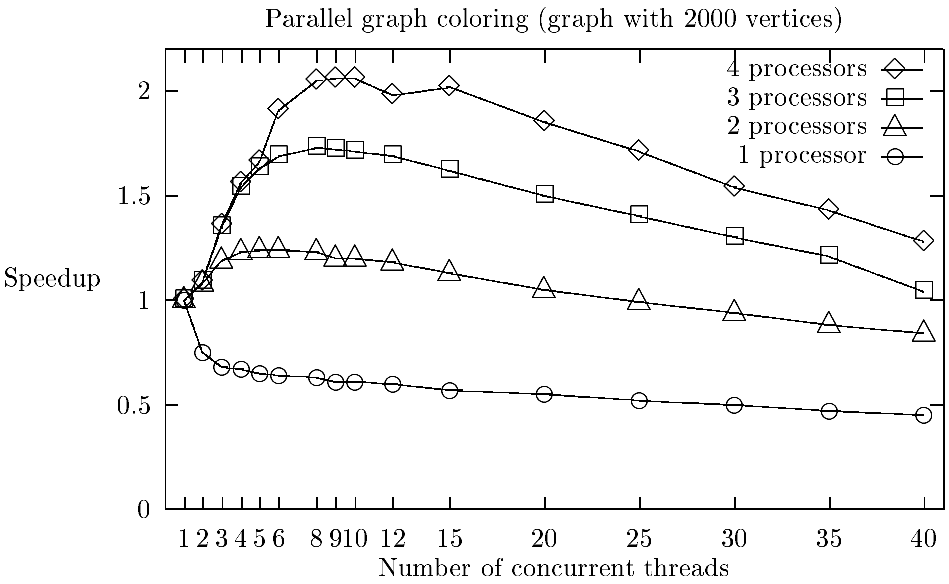

Figure 5: Speedup of the parallel algorithm using 1–4 processors.

processors are available. In section 3.3 was mentioned that the maximum speedup of this algorithm is restricted to only about half the number of processors. If we take this fact into account the achieved speedup is quite good.

5 Conclusions

In this paper we showed how an algorithm which seems to be inherently sequential could be converted to a parallel one. The resulting pipeline structure could be implemented easily in JAVA. The runtime and speedup results have been quite satisfactory. For future work it would be interesting to benchmark the implementation on a machine with more than four processors in order to test whether even more processors could be utilized or to see when the impact of the shared memory architecture prevents a further increase of the speedup.

References

[1] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[2] INMOS Limited. *Occam 2 Reference Manual.* Prentice Hall International, Englewood Cliffs, 1988.

[3] H. T. Kung. The structure of parallel algorithms. In M. C. Yovits, editor, *Advances in Computers*, volume 19, pages 64–112, New York, 1980. Academic Press.

[4] David W. Matula, George Marble, and Joel D. Isaacson. Graph coloring algorithms. In Ronald C. Read, editor, *Graph Theory and Computing*, pages 109–122, New York, 1972. Academic Press.

[5] T. Umland, *Über heuristische Verfahren zur Lösung des Färbungsproblems*, Ph. D. Thesis, University of Karlsruhe, VDI-Verlag, Düsseldorf, 1996.

[6] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10:85–86, 1967.