ACHIEVING EFFICIENT REGISTER ALLOCATION VIA PARALLELISM

Christine Makowski and Lori L. Pollock
University of Delaware

Keywords: Parallel register allocation, graph coloring, clique separators.

Introduction

Effective register allocation has become an important component of optimizing compilers. Unfortunately, it is also a very time-consuming task, one of the culprits of long compilation times. Multiprocessor parallelism is being recognized as the key to major improvements in program performance. In this paper, we describe how we exploit parallelism to increase the efficiency of register allocation without sacrificing the quality of the generated code. To our knowledge, this is the first parallelization of register allocation and experimental investigation into parallel graph coloring for register allocation.

Several researchers have recognized that global register allocation via graph coloring can be accomplished by decomposing the program via clique separators, coloring the segments independently, and then recombining the colorings to obtain an overall allocation [8, 9, 16]. Based on a result by Tarjan concerning the colorability of graphs partitioned via clique separators [14], optimality of the overall allocation for straight line code is not compromised through this method. For code with branches, optimal allocation along one execution path can be obtained, while cleanup code may have to be inserted along the other paths because the individually colored subgraphs do not fit together.

Gupta and Soffa [8] developed a sequential global register allocator based on clique separators with the goal of reducing the space and time requirements by coloring smaller portions of the program separately. Their experiments demonstrated that the space requirements for the interference graph did indeed decrease; their recent timings of the sequential clique separator approach showed that it increased the allocation time by 34% to 96% over the Chaitin allocation method [9]. However, as they noted, this clique separator approach to partitioning the interference graph also appears to be a promising platform for parallelizing the register allocation process.

Since the clique separators are used to determine the partitioning of the register allocation problem, the parallelism that can be achieved will be limited by the number of clique separators in the register interference graph. Furthermore, the quality of the resulting allocation will be affected by the handling of branches, due to the cleanup code that is sometimes inserted. Zobel's work [15, 16] focused on developing a preprocessor to register allocation to transform as much of the register interference graph as possible into interval graphs, which correspond to straight line code, in order to avoid cleanup code insertion. She was able to map 33 of her 34 benchmark interference graphs into interval graphs, and she observed an abundance of clique separators in well structured programs. These results suggest that there is considerable flexibility for the parallelization of global register allocation via this approach.

In this paper, we describe our efforts in developing a parallel register allocator based on the notion of clique separators, and our investigation into the performance that can be attained through parallelization. We have developed a parallel algorithm for register allocation with several communication schemes. We implemented the parallel register allocator in C-Linda [4], and performed experimental studies on a set of benchmark programs running on a Sequent Symmetry with 8 processors. Our parallel register allocator is currently designed to perform parallel coloring of interval graphs as we have been interested primarily in determining whether the parallelization of register allocation is worthwhile. Our experimental studies show that parallelization of register allocation results in significant decreases in allocation time, and furthermore, the performance gains improve with increasing program sizes. The quality of the allocations improved with increasing amounts of partitioning. This parallel allocator forms the basis for a parallel allocator which will input code with branches, and handle merging allocations for alternative paths and inserting cleanup code.

We begin with a brief background of register allocation via graph coloring and clique separators, and a closer look at the work by Gupta and Soffa [8]. Detailed descriptions of our parallel algorithm, the communication schemes, and rules for combining allocations are then presented. We describe our implementation, followed by a discussion of our experimental framework and results, and finally conclusions and future work are presented.

Background and Previous Work

Global register allocation is commonly based on graph coloring [5, 6, 7, 11, 1, 2, 3, 12]. An interference graph is constructed in which the nodes represent candidates for physical registers, and edges connect nodes that must be assigned different physical registers because their values will coexist during program execution. In particular, each node represents a live range where a live range is defined to be the set of definitions and uses where the definitions are all connected by common uses.

Using no more than k colors where k is the number of available physical registers, the allocator attempts to find a k-coloring of the interference graph. Interference graphs for straight line code are interval graphs. An interval graph is a graph in which the nodes can be represented by intervals of the real line such that two nodes are connected by an edge if and only if the corresponding intervals overlap. Interval graphs can be colored optimally in polynomial time, whereas finding an optimal coloring for an arbitrary interference graph representing a program segment with

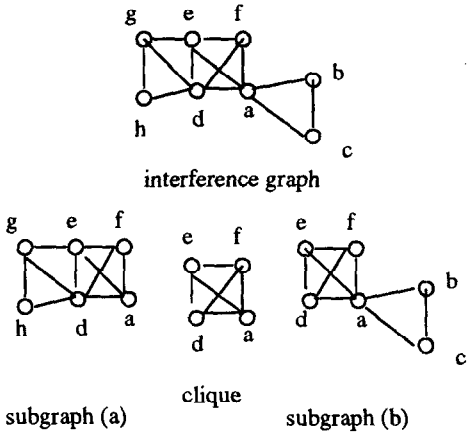

Figure 1: Interference Graph and Clique Separator

branches and loops is NP-complete [13]. Various heuristics have been developed for determining which order to color the nodes and which nodes to spill (i.e., which variables should be stored and loaded from memory) when there are not enough available registers.

A clique separator is a completely connected subgraph which when removed, disconnects the original graph into at least two separate subgraphs, which can be further decomposed if desired. Tarjan showed that if each subgraph can be colored with at most k colors, then the original graph can be colored using k colors by combining the colorings of the subgraphs [14]. In terms of register allocation, the entire interference graph can be partitioned by first identifying clique separators. Individual interference subgraphs are created by including the clique separator responsible for the partition in each of the disconnected subgraphs. Figure 1 shows an interference graph, a clique separator, and the resulting subgraphs. Each of the subgraphs can be colored independently. The colorings are combined to obtain a coloring for the entire interference graph (i.e., an allocation for the entire program segment) by recoloring the nodes in the subgraphs involved with each clique separator such that the disconnected subgraphs use the same colors for the nodes of the clique.

Gupta and Soffa [8] presented an algorithm for identifying clique separators directly from a program code segment. The subgraphs of the interference graph can be built and colored without requiring space for the entire interference graph at any given point in time. They first identify clique separators throughout the entire program, independently finding separators for each path that leaves a branch statement. The algorithm then loops sequentially over each partition to build the interference subgraph, color the graph, and then recombine its colors with previously processed partitions.

In straight line code, represented by an interval graph, each statement can be used to represent a clique because there are overlapping intervals at each statement. Furthermore, each of these cliques is a separator because removing the clique will create a set of intervals that end before that clique and a set of intervals that begin after that clique. There would be no edges between nodes represented by these two sets of intervals, as the first set does not overlap with the second set. Thus, any statement in a straight line code segment can be used as a

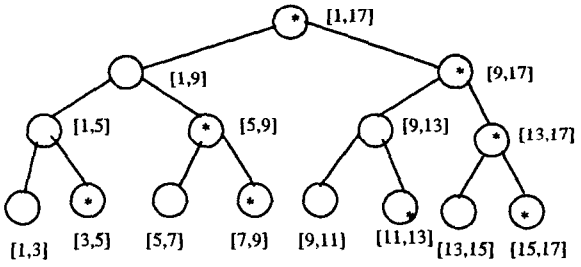

Figure 2: Tree Parallelism

clique separator. This flexibility is very desirable for parallelizing register allocation.

Parallel Coloring

We found that the development of an effective parallel algorithm for graph coloring based register allocation is not a straightforward parallelization of the Gupta and Soffa [8] sequential algorithm for register allocation via clique separators. First, the choice of which cliques should be used as separators has different implications when parallelism is being exploited. The technique that Gupta and Soffa use to select the clique separators sequentializes the processing of the program, whereas a method that allows clique separators to be chosen in parallel is desirable.

Second, because each subgraph is being colored in parallel, and not in a predetermined order, any assumptions about nodes that appear in a clique separator should carefully consider the introduction of unwanted dependencies between the processing of subgraphs which would inhibit parallelism. In particular, when partitions are colored sequentially, spilling and color assignments to nodes in the clique can be propagated to the next partition, avoiding an explicit recoloring when graphs are recombined. Unfortunately, this propagation would destroy all hopes of effective parallel coloring. Instead, in order to maximize the parallelism, it is desirable to color each subgraph independently and include an explicit step for recombining the separate colorings, which can be performed simultaneously with other subgraph colorings and recombining steps. This approach raises the new problem of how to combine the colorings of two separate subgraphs in order to obtain the coloring of their combined graph.

Third, in parallel processing, one always has to be concerned with trying to limit the amount of data that has to be communicated between processes. Otherwise, the communication overhead can destroy the potential for performance gains due to parallelism. Thus, the parallelization must carefully consider minimizing communication overhead.

In summary, our parallel solution differs from the Gupta and Soffa sequential clique separator solution in the following ways: (1) Our approach uses a parallel algorithm. (2) Due to the parallel approach, we must address the issue of minimizing communication without sacrificing parallelization. (3) We choose the clique separators in a different manner. (4) Because we color each subgraph independently of the colorings of other graphs, we have an explicit step for recombining the colorings of subgraphs to obtain a coloring for their combined graph.

```
Algorithm parcolor (codesegment,currdepth, desireddepth)
  if (currdepth == desireddepth)
  then
    seqcolor(codesegment)
    return results to parent process
  else
    /* create the new process */
    startnewprocess(parcolor(right half,currdepth+1,desireddepth))
    /* keep the left half in this process */
    parcolor(left half,currdepth+1,desireddepth)
    combineresults(left half, right half)
    return results to parent process
  endif
end parcolor

Program main()
  rename(program) /*renaming*/
  analyze(program) /* reaching definitions and live variables*/
  startnewprocess (parcolor(program, 0, desired depth))
  getresults(results)
end main
```

Figure 3: Parallel Coloring Algorithm

The Parallel Algorithm

Our parallel algorithm is based on *tree parallelism*. The straight line code segment is first partitioned into two equal-sized (or nearly equal-sized) subsegments, each consisting of a sequence of consecutive statements in the program. Each of these subsegments can be partitioned further into two equal-sized subsegments, and this binary partitioning process can be performed recursively until the desired level of parallelism is achieved. Thus, the division of the original code segment proceeds in powers of 2. This partitioning scheme takes advantage of the flexibility of every statement in a straight line code segment being a candidate for a clique separator.

Each time that the code is partitioned, one of the subsegments, call it the "right" half, is passed to a newly created process for allocation, while the "left" half of the segment is retained for allocation by the parent process, becoming a figurative left child. The eventual number of processes to accomplish the entire allocation will be the number of leaves in the binary tree, or $2^{desireddepth}$, where *desireddepth* is the desired number of levels of the process tree. Figure 2 shows the parallel process structure of the parallel algorithm for a program divided into 17 code blocks (i.e., sets of statements for program analysis; see Section 4.1) and 8 processes. In Figure 2, the bracketed pairs indicate the first and last code block of the set of code blocks assigned to each process. The nodes with asterisks represent newly created processes.

At the leaves of the process tree, a sequential graph coloring register allocation algorithm is applied independently and in parallel on each of the code subsegments. As each leaf process completes its coloring, it passes the result to its parent process. The parent process combines the colorings of its two children to create a coloring of the interference graph representing the combination of its children's interference graphs, and then passes its results to its parent. This process of combining graphs continues until the root process combines its children's graphs to form the complete colored interference graph. The parallel coloring algorithm is presented in Figure 3.

The parallel allocator allows any sequential graph coloring algorithm to be used on the individual code segments. The sequential coloring algorithm that we used for each subsegment follows the Chaitin method [5, 6]. Each subsegment is colored with the number of available colors equal to the number of available physical registers. Like the enhanced version of Chaitin's algorithm developed by Briggs, Cooper, and Torczon [1], we save the actual coloring of the nodes until the last step. In many cases, this will allow a node to be colored that would not have otherwise received a color.

Combining Allocations

The clique separator used for partitioning a code segment into two subsegments will appear in both of the resulting subgraphs. After coloring both subgraphs independently, the goal is to change the coloring of the right subgraph to match the coloring of the left subgraph. We call this process *recoloring*. There are four conditions that could hold for a given node that appears in both subgraphs because it is in the clique separator. (1) The node could be colored in both graphs, but possibly not the same color. (2) The node could be colored in the left subgraph, but not in the right subgraph, denoting that it was selected to be spilled in the right subgraph. (3) The node could be colored in the right subgraph, but not in the left subgraph, denoting that it was chosen to be spilled in the left subgraph. (4) The node was spilled in both subgraphs. Figure 4 gives the algorithm for combining the colorings of two subgraphs, called left and right. Figure 4 presents the recoloring process as a fairly straightforward process, but in fact, when we attempt to reduce the communication costs among the processes as described in the next section, implementing this recoloring algorithm becomes much more challenging.

All allocation is based on local, not global information. When a clique node is colored in one subgraph and not colored in the other subgraph, we say that the live range represented by that node is split at the boundary of the clique code block. In case (3), a load is inserted before each use of that variable in the left subgraph, as in traditional register allocation. In case (2), a store is inserted at the end of the leaf for the left subgraph if the variable is defined in that leaf and live at that point. Loads are inserted before each use of that variable in the leaf where the variable was said to be spilled in the right subgraph.

Communication Schemes

Although we have implemented our parallel algorithm on a shared memory machine, we have designed our algorithm with concern for communication costs between processes, as our goal is to port this software to a network of workstations. Thus, our algorithm does not rely on the underlying shared memory machine. In order to perform the allocation for each local code segment, each process will be passed its subsegment from the parent process in the binary process tree. Each subsegment then needs to communicate its results from allocation of its subsegment back to its parent process when it is finished, in order for the parent to combine the colorings of its left and right child processes.

One method of communicating results back up the binary process tree is to pass the colored interference subgraphs through each level of the tree. As the right child process passes its colored subgraph to its parent, the parent process renames the colors of the clique separator in this subgraph to match the colors of the figurative left child's subgraph. These color changes are then propagated throughout the entire right subgraph. Then, the parent combines the subgraphs from the left and right children, and passes this combined graph to its parent process. We call this method of communicating colored subgraphs the *graph com-*

```
Algorithm recoloring (left_subgraph,right_subgraph)
  Let clique=nodes in clique separator of left & right subgraph;
  for each node n in clique do {INITIALIZATION}
    Set needcolor(n) = FALSE;
  for each color c do
    Add c to available_color_bag;
    Set changed(c) = FALSE;
{RECOLORING - logically, we recolor individual nodes, for low}
{communication, we do this with only color map & clique nodes}
  for each node n in clique do
    Let lcolor = color of n in left_subgraph;
    Let rcolor = color of n in right_subgraph;
    if (lcolor <> spill) and (rcolor <> spill) then
      Replace rcolor by lcolor throughout right_subgraph;
      Remove lcolor from available_color_bag;
      Set changed(rcolor) = TRUE;
  endfor
  for each node n in clique do
    Let lcolor = color of n in left_subgraph;
    Let rcolor = color of n in right_subgraph;
    if (lcolor == spill) and (rcolor <> spill) then
      Record that n is split;
      Set needcolor(n) = TRUE;
    if (lcolor <> spill) and (rcolor == spill) then
      Record that n is split;
  endfor
  for each node n in clique do
    if needcolor(n) then
      Replace color of n throughout right_subgraph by a color c
        in available_color_bag not equal to current rcolor of n;
      Set changed(rcolor of n) = TRUE;
      Remove c from available_color_bag;
  endfor
  for each color c do
    if (changed(c) == FALSE) then
      Replace c throughout right_subgraph by a color d
        in available_color_bag;
      Set changed(c) = TRUE;
      Remove d from available_color_bag;
  endfor
end recoloring
```

Figure 4: Recoloring Algorithm

munication scheme. The downfall of this approach is that the amount of communication is proportional to the size of the program. Thus, for large programs, where one would hope parallel register allocation would be a big win over sequential allocation, communication overhead may outweigh the advantages of parallelism.

We developed an alternative method for communicating results from the allocations of the subsegments called the *color map communication* scheme. This scheme evolved in several steps. First, we decided to group the nodes by colors into sets, and name each set by its associated color. Then, as the graph is passed up the process tree, rather than explicitly recoloring each of the interference graph nodes that requires recoloring, we rename the sets representing the colors that need to be changed to a new color. In this way, the recoloring is proportional to the number of colors. However, the interference graph is still being passed at each level of the tree. Thus, although there is less work done to accomplish the recoloring step, the amount of communication is still proportional to the size of the program.

Since individual nodes are no longer being renamed at each level of the tree, it is not necessary to communicate them all to each process. Based on this observation, the final color map communication scheme is to pass only the color changes along with the clique nodes at each level. More specifically, each leaf process communicates its clique nodes with colors and a color change map to its parent process. Each leaf process also communicates the original subgraph nodes and colors (by bit vector) directly to the master process (i.e., *main*) where the final color assignments will be made. This is the only communication of the interference graph in this communication scheme.

At each level in the process tree, the interior node process receives the colors of the clique from its left and right children, and follows the rules for matching the right child's colors with the left child's color. It then receives a color change map from each leaf that is a descendant of its right child. With the current colors of the clique nodes, it makes the new changes in each of these leaves' color maps and passes them up the tree. Finally, in preparation for recoloring at the next higher level, the interior node process changes the colors of the nodes that will be in the clique at the next level to match the new color map.

As soon as the final color map for a particular leaf becomes available to the master process, it executes the following loop for that leaf process:

```
for each color[i]
  for each node[n] originally colored i
      finalcolor[n] = newcolor[i]
```

The *color map communication* scheme communicates only the color map and clique at each level. The clique size for each recoloring is not proportional to program size, but rather depends on the nature of interferences in the interference graph, which is a result of the pattern of variable uses and definitions. In our experiments, we found that in all test cases, the clique size is less than 16 nodes, and usually less than 10. Thus, the *color map communication* scheme provides communication of results effectively proportional to the number of colors, or physical registers, rather than the program size. This has two benefits over *graph communication*. The number of physical registers is typically a small relative to the number of variables in a program. And, as program size increases, the communication overhead will not increase proportionately, and thus much better speedups can be expected for larger programs.

Experimental Framework

The parallel allocator is implemented in C-Linda [4]. The experiments were performed on a Sequent Symmetry machine with 8 processors. The parallel allocator takes input code written in a low level intermediate code which assumes an unlimited number of virtual registers. It also takes as input the desired number of physical registers. These virtual registers are mapped onto physical registers, generating the final coloring of the entire interference graph along with the total number of spills. Variable renaming is performed first in order to reduce the number of definitions and uses in each live range. An analysis phase consists of computing reaching definition and live variable information. Output of this analysis phase consists of the set of live ranges in each code block. PARCOLOR takes this live range information, the number of available colors (registers), and the user defined degree of parallelism as input. As we have been interested in the effect of parallelization on performance, our timings are taken for the PARCOLOR phase only.

In order to work with straight line code, large basic blocks were sampled from a number of routines in the Riceps benchmark suite developed at Rice University. Routine sizes varied from 500 to 1500 statements with the number of live ranges in the routines varying from 481 to 1414.

Register interferences can be computed on a statement by state-

ment basis, or on a basic block basis. Chow and Hennessy [7] note that the statement by statement basis will give better allocations as the live ranges are more accurate; however, this approach is more costly to compute than interferences based on basic blocks. Because we are dealing with straight line code, we can experiment with different sizes of code blocks to observe the effects of this localization, and also how much the execution time of the parallel coloring approach degrades as the size of a code block approaches a single statement. Note that we use the term code block, as the whole code segment that we are using is a single basic block, which we are dividing into code blocks of various sizes for live range analysis. We experimented with dividing the code into code blocks of size 1, 4, 10, and 25 statements. Renaming, reaching definitions, and live variable analysis were all performed to gather live range information per code block.

Experiments were run with 8 and 16 physical registers (available colors). Each routine was colored with 1, 2, 4 and 8 processes. That is, the desired depth of parallelism was varied between 1, 2, and 3 levels of the tree parallelism. Consistent degradation of performance with more than 8 processes (more than 3 levels) was interpreted as reaching the limits of parallelism of our machine.

We compared the performance of our parallel algorithm (PARCOLOR) running with 2, 4, and 8 processes (recursively partitioning to 3 levels) against the SEQ-CLIQUE, sequential clique separator based register allocation (implemented by recursive tree partitioning and recoloring) and SEQ-TRADITIONAL, sequential register allocation via graph coloring with no partitioning (implemented by running PARCOLOR with 1 process and no partitioning). We measured the running time of SEQ-CLIQUE with 1, 2, 4, and 8 processes. That is, we measured breaking up the program into 1, 2, 4, and 8 parts and coloring and combining the colorings of these parts sequentially to determine the effect due to parallel processing versus the effect due to splitting up the program into smaller chunks. When we measured the time to set up the data structures for parallelization of the coloring process, we found this overhead to be 5 % of the total time to execute the allocator as 1 process. The following section presents our results.

Experimental Results

The *graph communication* method produced early promising results. For small programs with 2 or 4 processes, speedup curves for SEQ-TRADITIONAL versus PARCOLOR followed near linear increases. However, as program size increased, the performance degraded beyond 4 processes, and the percent degradation increased as the programs increased in size. This was a reflection of the increased load on the communication system (which is proportional to graph size) beginning to outweigh the benefits of parallelism. There is no way to reduce these costs using the graph communication scheme. Thus, this method is practical only for small programs.

Figure 5 depicts the ratio of (Running Time(SEQ-TRADITIONAL) / Running Time(PARCOLOR)) for 8 physical registers and code block size of 10 statements. The labels on the lines indicate the routine from which the large code blocks were taken. The dotted line indicates linear speedup. Speed improvements of PARCOLOR over SEQ-TRADITIONAL in all cases were excellent. The allocator ran all routines at least twice as fast with 2 processes as with one, 4 to 6 times faster with 4 processes, and 5 to 12 times faster with 8 processes. The number of available registers and the code block sizes had little effect on the performance. The speed improvements held true for both

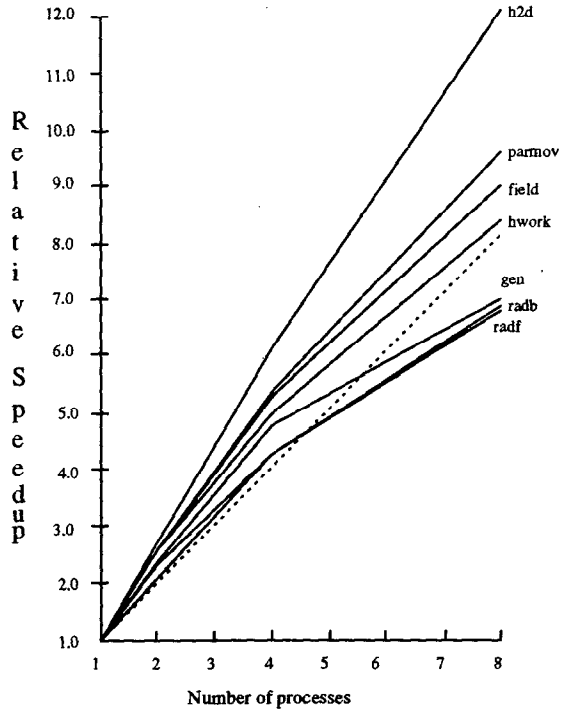

Figure 5: Parallel versus Non-partitioned Sequential Coloring

8 and 16 registers at all levels of parallelism and all values of code block sizes. Not only was the speedup superlinear, but this performance ratio improved as the input data size increased.

The speed improvement curves for each routine reflect the fact that a large percentage of the time incurred in register allocation is involved in building and coloring the interference graph. The clique separator method produces potential for speed improvements in two ways. First, it divides the program into independent parts which can be processed in parallel. We call this the *parallelization effect*. Second, it is expected that the sizes of the subgraphs being colored by each process will be significantly smaller than the original interference graph. Since graph coloring time is polynomial in the number of nodes in the interference graph, the total amount of graph coloring time should be less even in a sequential processing of the segments. Each node is now concerned only with its interference with the nodes in its local process, rather than nodes over the entire graph. We call this the *clique separator effect*.

Our results also showed that the higher speedup values in all cases are associated with the larger programs. Instead of degradation in performance as the data size increased, we were observing an improvement. We believe that this is because the communication pattern of the algorithm is not related to the size of the input, but remains almost constant (proportional to the number of registers) for all input program sizes.

The purpose of our next experiment was to determine the part of the performance improvement attributable to the clique separator effect versus the parallelization effect. First, we studied the speedup of the clique separator algorithm run on one processor using tree division and recombination. This is the algorithm that we call SEQ-CLIQUE. The input graph was divided into

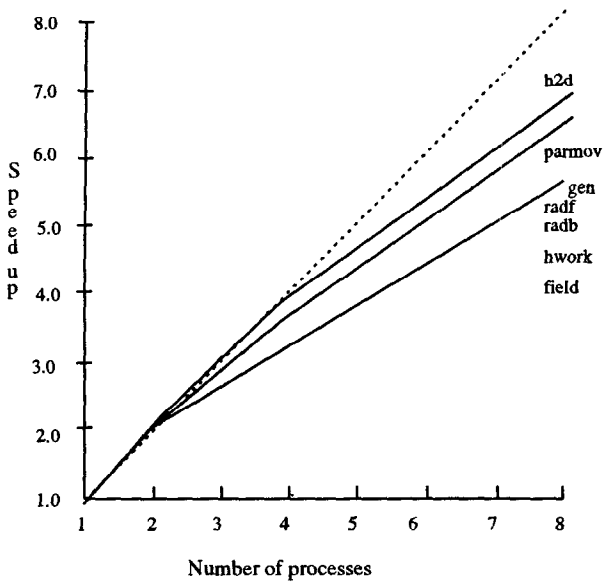

Figure 6: Parallel versus Sequential Partitioned Coloring

2, 4, and 8 parts to correspond to the division scheme in PARCOLOR. After subtracting the time for data structure setup and communication, speedup was calculated as (Running Time(SEQ-TRADITIONAL) / Running Time(SEQ-CLIQUE with N parts)) with values of 2, 4, and 8 for N. The results were very similar for all test cases. The speedup for all values of 2, 4, and 8 parts remained constant at approximately 1.8.

Next, to calculate the speedup due to parallelism, we calculated (Running Time(SEQ-CLIQUE with N parts) / Running Time(PARCOLOR with N processes)). That is, for example, the time for SEQ-CLIQUE with 4 parts is divided by the time for PARCOLOR with 4 processes. Figure 6 shows these speedup results. The speedup from parallelism is near linear due to the low communication costs of the color map communication scheme. By looking at this graph and from our speedup results of SEQ-CLIQUE, we conclude that beyond division into two parts, the major contributor to the overall speedups is parallelism.

We experimentally compared the quality of allocations by PARCOLOR, SEQ-TRADITIONAL, and LOCAL, a local allocator based on the algorithm by Hsu, Fischer, and Goodman [10]. Note that SEQ-CLIQUE and PARCOLOR will give the same allocations for the same partitioning, so these results are given as PARCOLOR. Quality of allocation comparisons were made by counting the total number of variable uses spilled over the entire graph for each parallel coloring scheme. Counting the number of uses is sufficient because each live range will have exactly one definition due to renaming, and straight line code eliminates the need to deal with estimating the frequency of loop executions.

In about 50% of the cases, PARCOLOR was comparable to (within 10% of) or better than LOCAL. In addition, PARCOLOR ran in half the time. Because our current parallel allocator is restricted to interval graphs, we expected that the LOCAL allocator would generate better allocations. In all cases, PARCOLOR gave better allocations than SEQ-TRADITIONAL, demonstrating that the clique separator partitioning achieves a localization that improves graph coloring allocation.

In addition, we examined the effects of increased localization in construction of the interference graph by varying the size of the code blocks in the routines. Since interferences are computed on the code block level, decreasing the size of the blocks should create fewer interferences between nodes. For the experimental runs, each routine was divided into four code block sizes: 1, 4, 10, and 25 statements per block. We expected to see a decrease in the number of spills as the interference information becomes more localized. This is relevant in exploring the possibilities of further dividing basic blocks after using the preprocessing scheme of Zobel to create straight line code.

For each routine, as the code block size decreases, the allocation improves (i.e., the total number of spills decreases). This pattern was observed running as 1 process (SEQ-TRADITIONAL), and at all 3 levels of parallelism. Across all runs of all the routines, the best allocation occurred when combining the minimum code block level with the maximum degree of partitioning. A very important observation is that statement level allocation drastically reduces the number of spills without significant loss in performance. Also, as expected, by increasing the number of available registers, a significant reduction in the number of total spills can be achieved without any degradation in algorithm performance.

Conclusions

The primary contribution of this paper is the development, implementation, and empirical investigation of a parallel graph coloring register allocator. One advantage of our parallel algorithm is that we believe that it allows any method for coloring the interference graph of each subsegment. Of particular interest is the fact that our parallel coloring algorithm achieved very good speedups over the sequential graph coloring allocator. When we accounted for the clique separator effect, we found that most of our speedup is due to parallelism, not to the partitioning of the graph by clique separators. In addition, we found that the performance gains continue to improve with larger program sizes, and the quality of the allocations increase with an increasing amount of partitioning.

Our experimental results have been very encouraging. The good speedups and the increasingly larger speedups as the program size grows provide the impetus to continue our work in several directions: (1) extending the parallel allocator to handle code with branches, and (2) exploring more expensive, but more precise, allocation heuristics, and (3) exploring the application of this work to other graph coloring problems.

Authors' email addresses: makowski@udel.edu, pollock@udel.edu.

References

[1] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, July 1989.

[2] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation*, June 1992.

[3] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 192--203, June 1991.

[4] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs*. MIT Press, 1991.

[5] G. J. Chaitin. Register allocation and spilling via graph coloring. In *ACM SIGPLAN Symposium on Compiler Construction*, June 1982.

[6] Gregory Chaitin, Marc Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47--57, January 1981.

[7] Frederick Chow and John Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12:501--536, October 1990.

[8] R. Gupta and M. L. Soffa. Register allocation via clique separators. In *ACM SIGPLAN Programming Language Design and Implementation Conference*, pages 264--274, 1989.

[9] Rajiv Gupta, Mary Lou Soffa, and Denise Ombres. Efficient register allocation via coloring using clique separators. *ACM Transactions on Programming Languages and Systems*, 16(3):370--386, May 1994.

[10] W. Hsu, C. Fischer, and J. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Transactions on Software Engineering*, 15(10), October 1989.

[11] J. R. Larus and P. N. Hilfinger. Register allocation in the spur lisp compiler. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 255--263, 1986.

[12] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 300--310, 1992.

[13] Ravi Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4:226--248, 1975.

[14] R.E. Tarjan. Decomposition by clique separators. *Discrete Mathematics*, 55:221--231, 1985.

[15] Angelika Zobel. *Program Structure as a Basis for the Parallelization of Global Compiler Optimizations*. PhD thesis, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992.

[16] Angelika Zobel. Program structure as basis for parallelizing global register allocation. In *International Conference on Computer Languages*, 1992.