THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

A PERFORMANCE ANALYSIS OF PARALLEL MULTIGRID COARSENING

By

JUSTIN A. SLONE

An Honors Thesis submitted to the
Department of Computer Science

Degree Awarded:
Spring Semester, 2004

The members of the Committee approve the dissertation of Justin A. Slone defended on March 19th, 2004.

Kyle Gallivan
Professor Directing Thesis

Robert van Engelen
Committee Member

Anuj Srivastava
Outside Committee Member

To Rachel

ACKNOWLEDGEMENTS

Throughout the course of this thesis I have had the privilege of working with top rate professors. My directing professor, Dr. Kyle Gallivan, has been a valuable mentor and colleague. One who, in this last year, has taught me the meaning of rigor. I would also like to thank my committee members Robert van Engelen, and Anuj Srivastava. Finally, I extend special thanks to Chris Baker who has been an invaluable asset helping me wade through the strange world of parallel performance analysis.

TABLE OF CONTENTS

List of Tables . . . . . **vii**

List of Figures . . . . . **viii**

Abstract . . . . . **x**

1. INTRODUCTION **1**

1.1 Multigrid 2
1.2 BoomerAMG 3

2. COARSENING **5**

2.1 Basic Definitions 5
2.2 RS Coarsening 6
2.3 The CLJP Algorithm 7
2.3.1 Coarse point selection 7
2.3.2 Updating edges and the weights of neighbors 8
2.3.3 Edge removal 10

3. THREE BASIC VERSIONS **15**

3.1 Row-oriented version 15
3.2 Column-oriented version 18
3.3 A Hybrid Version 21
3.4 Implementation 24
3.4.1 Experiments 24
3.4.2 Membership in G_i 27
3.4.3 Edge Storage 28
3.4.4 Edge Removal 29
3.4.5 Removal Effectiveness experimentation 29
3.4.6 Coarse-Fine Status 33
3.4.7 Intersections 36
3.4.8 Synchronization 36

4. MODIFICATIONS **39**

4.1 Compressed Sparse Row 39
4.1.1 Data structure definition 39
4.1.2 Level of Parallelism 40
4.1.3 Parallelization 43
4.1.4 Node removal 46
4.1.5 CSR Edge Removal 48

4.1.6 Weight updates . . . 51
4.1.7 Conclusion . . . 53
4.2 Compressed Sparse Column . . . 55
4.2.1 Data structure definition . . . 56
4.2.2 Similarities between the CSR and CSC algorithms . . . 56
4.2.3 Differences between the CSR and CSC algorithms . . . 58
4.3 Summary of the CSR and CSC algorithms . . . 60
4.4 Modified Version . . . 61
4.4.1 Similarities between CSR, CSC, and MOD . . . 62
4.4.2 Differences between all three algorithms . . . 63
4.4.3 MOD algorithm performance . . . 63

5. CONCLUSIONS . . . 67

5.1 Summary . . . 67
5.2 Conclusion . . . 71
5.3 Future Work . . . 73

REFERENCES . . . 75

BIOGRAPHICAL SKETCH . . . 76

LIST OF TABLES

3.1 Times for various methods of synchronization (average of 10 runs) 38

LIST OF FIGURES

2.1 CLJP Weight and Edge Update Algorithm 10
2.2 Edge updates 11
3.1 Row-oriented Pass 1 based on similar algorithm in [4] 17
3.2 Row-oriented Pass 2 based on similar algorithm in [4] 18
3.3 Column-oriented Pass 1 based on similar algorithm in [4] 19
3.4 Column-oriented Pass 2 based on similar algorithm in [4] 20
3.5 Hybrid version Pass 1 based on similar algorithm in [4] 22
3.6 Hybrid updates 22
3.7 Hybrid Pass 2 based on similar algorithm in [4] 23
3.8 Example of a (a) nine-point and (b) five-point stencil 25
3.9 Effectiveness of node and edge removal for the row-oriented implementation. . . 30
3.10 Effectiveness of node and edge removal for the column-oriented implementation. 31
3.11 Effectiveness of node and edge removal for the hybrid implementation. 32
3.12 The number of edge visits for each version without edge removal (logarithmic scale). Same instrumented statistics used in the measure of effectiveness (Figures 3.9-3.11) 33
3.13 Abbreviated row-oriented pass 1. With outer and inner loops marked. 34
3.14 The speed up for edge removal vs edge pruning 35
4.1 Speed up comparison for outer loop parallelism and inner loop parallelism 42
4.2 The bi-directional connections between a single node in a graph constructed with a nine point stencil and a value of 1 for K. 44
4.3 Speed up over granularity of 1 for varying lock granularities on the CSR algorithm. 10 processors, 1 million nodes, Ninepoint laplace operator 46
4.4 Speed up for processor local active list, for various list sizes, vs the base version without node removal. 47
4.5 Speed up for node and edge removal vs base implementation. Density: K=1 . . 49
4.6 Speed up for node and edge removal vs base implementation. Density: K=2 . . 50
4.7 Speed up for node and edge removal vs base implementation. Density: K=3 . . 51

4.8 Performance of the Weight Vector modification. Speed up for Weight Vector vs. the base implementation. 53

4.9 Performance of the Weight Vector^T modification. Speed up for Weight Vector^T vs. the base implementation. 54

4.10 Performance of Hash table modification. Speed up for the Hash table vs. the base implementation. 55

4.11 Speed up for Node removal (CSC). A simple comparison between the execution time of the CSC algorithm with and without node removal for a number of densities (K). This figure shows that node removal has no significant impact on the execution time of the CSC algorithm. 57

4.12 Improvement of the CSC algorithm over the CSR algorithm. 61

4.13 Speed up for Node removal (MOD). A simple comparison between the execution time of the MOD algorithm with and without node removal for a number of densities. This figure shows that node removal has no significant impact on the execution time of the MOD algorithm. 62

4.14 Increase in edge visits for the MOD algorithm vs the CSC algorithm. Number shown is how many times more edge visits MOD performs. 64

4.15 Improvement of the MOD and CSC algorithms over the CSR algorithm. Speed ups, $S(p, K, CSC)$ and $S(p, K, MOD)$, show how many times faster MOD and CSC is than CSR. 65

4.16 Improvement of the CSC algorithm. Speed up, $S(p, K, CvM)$, shows how many times faster CSC is than MOD. 66

5.1 True speed up for parallel CSR algorithm, compared to the sequential CSR algorithm (has no synchronization calls). 69

5.2 True speed up for parallel CSC algorithm, compared to the sequential CSR algorithm (has no synchronization calls). 70

5.3 True speed up for parallel MOD algorithm, compared to the sequential CSR algorithm (has no synchronization calls). 71

5.4 Parallel overhead, measured in speed up of the sequential version. 72

ABSTRACT

The solving of linear systems of equations is often one of the most time-consuming computational kernels of modern simulations. This thesis investigates a significant trade-off for a key portion of a state-of-the-art numerical library for solving very large sparse linear systems of equations based on a multigrid solver developed at Lawrence Livermore National Laboratory. It tests the hypothesis that the influence of the choice of data structure on synchronization is a major factor in the performance of the coarsening algorithm. It does this by examining three approaches to the CLJP multigrid coarsening algorithm from an analytical point of view and via careful incremental modification of implementations on an 11 processor shared memory machine (Sun Enterprise E4500).

CHAPTER 1

INTRODUCTION

Many scientific and engineering applications require some type of simulation, such as the simulation of car bodies or the effect of earthquakes on buildings. In these simulations, solving very large, sparse linear systems of equations is often one of the most time-consuming computational kernels. Therefore, any reduction in the time required to solve such systems, will have far-reaching benefits.

Two important characteristics of any algorithm used to solve very large sparse linear systems of equations, are computational complexity, measured in number of operations required to produce a solution, and accuracy. As the size of the problem increases, the computational complexity of the algorithm must be tractable for problem sizes of interest to the application scientist or engineer. For example, given an algorithm that requires $O(n^5)$ floating point operations, on an architecture that can perform 5 Gflops, a problem of size $n = 10^5$ will take over 2 years to solve and not be a viewed as a practical approach. In addition to reasonable computational complexity, it is also required that, when using finite precision arithmetic, the algorithm produces numerically reliable results i.e. a result with enough significant digits of accuracy for the application scientist. Given an algorithm with tractable computational complexity and desired accuracy, the implementation of this algorithm on a particular architecture must, in the time allowed, produce a valid result. To achieve this goal, trade-offs between complexity, accuracy, amount of storage, communication requirements, and number of processors must be considered.

This thesis investigates a significant trade-off for a key portion of a state-of-the-art numerical library for solving very large sparse linear systems of equations based on a multigrid solver developed at Lawrence Livermore National Laboratory. It tests the hypothesis that the influence of the choice of data structure on synchronization is a major factor in the performance of the coarsening algorithm. It does this by examining three approaches to

the CLJP multigrid coarsening algorithm from an analytical point of view and via careful incremental modification of implementations on an 11 processor shared memory machine (Sun Enterprise E4500).

1.1 Multigrid

Multigrid is a group of methods for solving very large sparse linear systems of equations. The principles of multigrid are explained in [1] and [7]. In general, multigrid can be divided into two classes, geometric multigrid, and algebraic multigrid.

Geometric multigrid uses a fixed hierarchy of length scales based on physical properties of the grid on which the equations are defined. It defines a series of progressively smaller length scales and associated spatial grids. It then iterates across this series and, by a process called relaxation, reduces the error on every grid in the series, thereby approximating the solution. Because of its heavy reliance on the physical properties of the grid, geometric multigrid requires a problem that provides a grid with a meaningful interpretation in the physical domain. This is a significant limitation and has driven much of the recent research into algebraic multigrid.

Development of algebraic multigrid (AMG) began in the 1980's when the Galerkin-based coarse-grid correction process was introduced into geometric multigrid. It was found that the Galerkin coarsening operator could be derived directly from the underlying matrices without reference to the grids. Although the Galerkin coarsening is still geometrically based, it led others to look into creating a multilevel method based purely on the underlying matrices[9]. This multilevel method became known as algebraic multigrid. AMG today, is still based on the same, highly developed, principles as geometric multigrid but, the relaxation and coarsening schemes are re-defined. In leaving the physical grid behind, AMG looses the fixed hierarchy of grids. This is replaced by a variable coarsening scheme, based around a fixed relaxation scheme. The fixed relaxation scheme defines a new *algebraic* sense of smoothness, without a reliance on any type of physical grid. AMG can be applied to a much broader range of problems, making it an enticing field of study.

1.2 BoomerAMG

BoomerAMG is a well-known AMG implementation, developed by Van Emden Henson, Ulrike Meier Yang and others at Lawrence Livermore National Laboratory. The details of the implementation are described in [5]. BoomerAMG is the most recent and complete piece of research done on parallel AMG libraries. It has been implemented and tested on problems involving tens of millions of unknowns running on architectures with over a thousand processors.

BoomerAMG employs a number of different coarsening strategies. The first strategy is the Cleary-Luby-Jones-Plassman (CLJP) parallel coarsening algorithm. This algorithm uses a modified parallel maximal independent set algorithm developed by M. Luby[6]. Unlike the rest of the strategies employed by BoomerAMG, this coarsening is not a modification of the classical sequential Ruge-Stüben (RS) algorithm [7]. Because of the use of an independent set algorithm, CLJP allows multiple processors to work in the same domain at the same time. This eliminates the need for domain partitioning and thus removes the problem of boundary conditions between processors.

The beginning of the line of RS-based strategies is the RS3 algorithm (Ruge-Stüben third pass coarsening). RS3 is the simplest type of parallel RS coarsening used in the implementation of BoomerAMG. It consists of splitting up the domain into sub-domains each of which is assigned to a processor. The classical RS algorithm is applied by each processor to its sub-domain. After these independent tasks are complete, a "third pass" is performed to modify the coarsenings on the boundaries between sub-domains. A significant disadvantage of this algorithm is that each processor can only coarsen down to a single grid point per sub-domain. When employing tens of thousands of processors, this leaves the final coarse grid containing tens of thousands of points. Ideally, the coarsening should continue with multiple processors cooperating on coarsening the grid formed by the union of their undetermined points rather than forceing it to be considered coarse because a single point remains on each processor.

The two other coarsening strategies available in BoomerAMG are hybrids of the RS and CLJP algorithms. In Falgout coarsening the classical RS algorithm is applied to the interior of the partitioned domain, then CLJP is applied to the already partially coarse grid. This

provides more RS like coarsening throughout the majority of the grid, while CLJP is used to handle the boundaries between the partitioned domains. BC-RS coarsening is similar to Falgout, except the order is reversed. In BC-RS coarsening, the boundaries are first coarsened with CLJP, then RS is used to coarsen the interiors. Experiments have shown that this strategy leads to worse overall performance than the Falgout and RS3 coarsenings [5].

BoomerAMG is designed to utilize a distributed memory architecture[3]. There is a significant potential for increased performance, by adapting BoomerAMG to a shared memory, or hybrid clustered architecture. This thesis implements and analyzes a shared memory parallel multigrid coarsening algorithm suitable for use in BoomerAMG.

CHAPTER 2

COARSENING

This chapter summarizes the two basic coarsening algorithms of interest in this thesis: RS coarsening and CLJP coarsening.

2.1 Basic Definitions

The concepts of *dependence* and *influence* used in AMG are defined in terms of the linear system of equations, $Ax = b$, to be solved. Node i is said to depend on node j if the value of the unknown x_j is important in determining the value of x_i using the i-th equation. Influence, is defined as the inverse of dependence, if node i depends on node j then node j influences node i.

The *influence matrix* S is defined as:

$$S_{ij} = \begin{cases} 1 \text{ if } j \in S_i \\ 0 \text{ otherwise} \end{cases} \tag{2.1}$$

Essentially, $S_{ij} = 1$ only if i depends on j. The matrix S then can be used to construct the directed graph, G, used by the CLJP algorithm. For every $S_{ij} = 1$ an edge is added from node i to node j in the graph G. At first this produces a symmetric graph, however, it is not a requirement of the algorithm that the graph maintain this property. The edges in G can be described from two different points of view. Any edge in G can either be defined as the *dependence* or *influence* between two nodes. In terms of the graph G, the edges leaving node i are said to be the dependencies of i and are contained in the set S_i. Conversely the edges ending at node i are the influences of i and are contained in the set S_i^T (note: $(S_i)^T \neq S_i^T$). The two ways of looking at the graph G (in terms of either dependence:S_i or influence:S_i^T) are exploited later in the discussion of row-oriented and column-oriented data structures.

2.2 RS Coarsening

The key element of AMG is the coarsening step where points on a given grid are selected to form a coarser grid. This is done recursively in order to define the finest-to-coarsest grid hierarchy. The term 'strongly connected' is used below, this refers to two points a and b, where there exists a connection from both a to b and b to a.

RS coarsening employs two rules:

(R1) Every point to which a fine point is strongly connected, should either be a coarse point, or should be strongly connected to a coarse point to which the fine point is connected.[7]

(R2) The coarse points should be selected such that the final coarsening has the maximum amount of coarse points where no two coarse points are strongly connected.[7]

It is important that R1 is enforced strictly since it is designed to insure that the final coarsened grid satisfies the requirements of the AMG interpolation operator. R1 also improves the convergence factor of each grid level, increasing overall efficiency.

While the coarse points added by R1 improve the convergence factor, too many redundant coarse points increases the amount of work required during relaxation. By using R2 as a guide, the final coarsening contains the minimum number of coarse points possible, without violating R1, and thus preserves the requirements of the interpolation operator.

Sequential RS AMG coarsening produces a grid hierarchy using two passes over the current grid at each level. In the first pass, a coarse point with a maximal weight is selected and all points that are strongly connected to that coarse point are set as fine points. For each new fine point, the weights of all neighboring points are increased. Also the weights of all points to which the initial coarse point is strongly connected, are decreased. The next coarse point is selected and the process repeats. This pass terminates when all nodes in the grid are either coarse or fine.

The second pass of RS insures that R1 is met, while trying to enforce R2 as strictly as possible. For every dependence between two fine points that do not depend on a common coarse point, the algorithm tentatively changes the second of those fine points to a coarse point. The rest of the dependencies of the first fine point (outgoing edges) are then checked.

If more fine-fine dependencies exist, the first fine point is changed into a coarse point, the second fine point reverts from coarse to fine, and the algorithm continues with other fine-fine pairs. This form of coarsening produces the *best* set of coarse grids. However, due to its sequential nature, it cannot, without modifications, take advantage of parallelism[5].

2.3 The CLJP Algorithm

Most of the code used in parallel AMG can be adapted from the work done on parallelizing geometric multigrid. However, a key step in the setup phase of AMG, the coarse-grid selection, is inherently sequential in nature, and requires a great deal of work to be parallelized properly. The coarsening method on which this thesis focuses is the CLJP algorithm.

The CLJP algorithm is a two-pass coarsening algorithm, however, the two passes are applied to a sequence of graphs G_i, $i = 1, \cdots$, each of which comprises the part of the original graph G that has not been assigned coarse or fine status as of step i of the algorithm. This is done by defining a coarse node subset of the current undetermined graph and updating the graph in response to that selecttion. After updating the algorithm repeats by examining the set of remaining undetermined nodes and defining the next coarse node subset. The key components of the algorithm are: the selection of the next set of coarse nodes from the current set of undetermined nodes; the update of the edges associated with the current set of undetermined nodes; and the selection of the fine nodes after the updates have been made.

2.3.1 Coarse point selection

The CLJP algorithm determines the set of coarse nodes in a manner that is parallel and naturally avoids ties. To begin the process, each node in the original graph is assigned a weight that is equal to the number of its incoming edges (its influences), or equivalently the count of nonzeros in the corresponding column of the matrix that defines the system to be solved. The CLJP algorithm modifies each of the weights by adding a random number between 0 and 1. This effectively breaks any ties between weights assigned to nodes and allows the definition of local maxima in the graph.

At each stage of the algorithm, the coarse nodes are selected to be the nodes that are local maxima of the weights. The set of neighbors that define the local set of v is the set of nodes, u, that are destinations of an edge (v, u) (an outgoing edge of v) and all those with v as a destination (an incoming edge of v). If no nodes are found whose weights are greater than v, v is selected to be in the next coarse set (C_{i+1}). In general, several sets of coarse nodes are determined in order to assign a state of coarse or fine to each of the N nodes in the original graph. These sets are denoted $C_i, i = 1, ..., k$.

2.3.2 Updating edges and the weights of neighbors

At each step of the CLJP algorithm the graph, G_i, is defined by the undetermined nodes, V_i, and edges E_i that remain after the processing of edges and coarse nodes in sets $C_h \mid h < i$ and the induced fine nodes $F_h \mid h < i$. For example, G_1 represents the original graph, $C_1 \subset V_1$, is the initial coarse set, $V_2 = V_1 - C_1 - F_1$, and $E_2 \subset E_1$ are the sets of nodes and edges respectively that result from the update of G_1 given C_1.

Once the new maximal independent set of coarse points, C_i is selected, the algorithm updates the weights of nodes that neighbor each new coarse node. This is done according to two conditions defined in [2].

C1: Values at coarse nodes are not interpolated; hence, neighbors that influence a coarse node are less valuable as potential coarse nodes themselves.[2]

C2: If k and j both depend on c, a given coarse node, and j influences k, then j is less valuable as a potential coarse node, since k can be interpolated from c.[2]

Of course, these conditions must be translated into graph updates. The CLJP algorithm updates edges in E_i based on the selection of C_i by removing all edges, $e = (p, v)$, that are one of the following three types:

- Type 1 edge has $p \in C_i$.
- Type 2 edge has $v \in C_i$.
- Type 3 edge is such that $\exists (p, u) \in E_i, (v, u) \in E_i$ such that $u \in C_i$.

Types 1 and 2 arise from condition C1 and Type 3 arises from condition C2. As the edges are removed, the weights associated with the destination node of each edge must be decremented in preparation for the determination of C_{i+1}. These three types of edges, combined with the conditions defined above, drive the removal of edges from the graph. However, some care must be taken when defining what "removing" an edge means and when it is performed. This definition is formalized by specifying a more rigorous form of the processing of G_i given C_i.

The process of creating G_{i+1}, given coarse set C_i can be defined by the following algorithm and definitions. R_i is defined as the set of edges that are removed from E_i at the end of the pass that processes G_i given C_i. That is, when identifying edges for removal from E_i by considering the types above, E_i is not updated immediately after finding an individual edge to be removed. Those in R_i may be safely removed from E_i after all edges in G_i have been considered for removal. Due to condition C2 there is also a set that must be maintained beyond processing G_i, i.e., that are removed from E_i but used in later identifications of edges of Type 3. The set Q is defined as the set of edges that have been removed from E_i by condition C2, i.e., Type 3 identification, but must be used to apply condition C2 in to identify Type 3 edges in subsequent passes processing $C_j \mid j > i$. The justification for these definitions are given in Section 2.3.3. Finally, for notational convenience, T_i is defined as $T_i = Q \cup E_i$.

The CLJP algorithm can be expressed abstractly as in Figure 2.1. The algorithm does not include the determination of the set of new fine points F_i. Formally, this is determined after removal of the edges and the update of the weights.

The set F_i is taken to be all nodes with an updated weight less than 1. This implies that all incoming edges to the node have been removed, i.e., the node no longer influences any other nodes. Note, however, that this does not imply that all outgoing edges have been removed and care must be taken to have any destination nodes of such edges processed correctly and eventually become coarse nodes. There are two basic ways to handle this, first, a potentially fine node could stay in G_i until all of its outgoing edges have been removed. Alternatively, the edge could be added a special set, which is processed separately from G_i. The choice that yields the most efficient implementation depends strongly on the particular implementation of the CLJP algorithm and its data structures.

for each $c \in C_i$
Type 1 edges. $\forall j \mid (c,j) \in E_i$
$w(j) \leftarrow w(j) - 1$
$R_i \leftarrow R_i \cup (c,j)$

Type 2 edges. $\forall j \mid (j,c) \in E_i$
$R_i \leftarrow R_i \cup (j,c)$

Type 3 edges. $T_i \leftarrow Q \cup E_i$
$\forall j \mid (j,c) \in T_i$
$\forall k \mid (k,j) \in E_i$
if $(k,c) \in T_i$
$w(j) \leftarrow w(j) - 1$
$Q \leftarrow Q \cup (k,j)$
$R_i \leftarrow R_i \cup (k,j)$
end for
$E_{i+1} \leftarrow E_i - R_i$

Figure 2.1. CLJP Weight and Edge Update Algorithm

The CLJP algorithm repeats the steps in Figure 2.1 until all nodes are defined as either coarse or fine. An example of the edges removed from a graph is given in Figure 2.2. The edges that are removed are slashed to indicate their type, i.e., the number of slashes is the edge type number. The nodes c_1 and c_2 are coarse nodes in C_i. Only edges (w,p) and (p,x) are retained in E_{i+1}.

2.3.3 Edge removal

In the form of the CLJP algorithm given in Figure 2.1 edges are not removed immediately but instead placed in the set R_i or Q. To show why this must be done this section discusses the earliest point in the CLJP algorithm each of the three types of edges can be removed from E_i without affecting correctness. In the following argument, it is shown that edges of Type 1 can be removed immediately, while edges of Type 2 must remain until the processing of C_i is complete, and G_{i+1} is determined. Edges of Type 3 must remain available until the end of processing in step $j > i$ during which their destination node is determined to be a

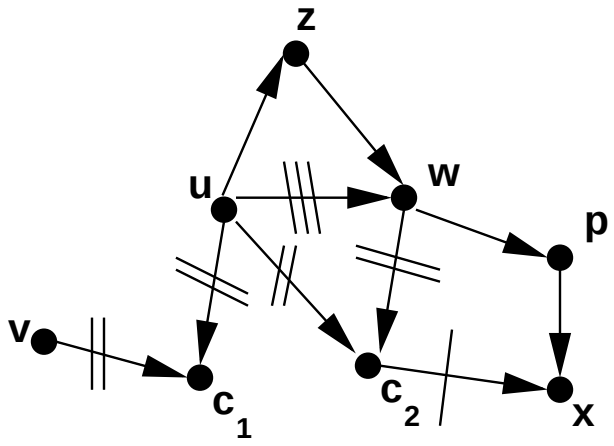

Figure 2.2. Edge updates

fine node, or the destination node is determined to be coarse, i.e., the Type 3 edge becomes a Type 2 edge. If an edge is first determined to be Type 3 on step i then it is added to Q and removed from E_i. Once the condition for final removal is met, but not before, the edge may be removed from Q.

First, consider the independent local maxima algorithm which produces C_i for pass i and note that, by definition, there cannot be an edge between any two nodes in the coarse set C_i. This part of the definition is used several times in the argument below.

Type 1 edges: An edge $e = (c, v)$ where $c \in C_i$ is a Type 1 edge, e.g., edge (c_2, x) in Figure 2.2. Suppose e is determined to be Type 1 on while processing C_i during step i of CLJP. Clearly, this edge cannot affect the detection of other Type 1 edges during step i. The edge e cannot also be of Type 2, because a Type 2 edge requires that the destination of the edge be in C_i. The source of this edge is already in C_i, and by the definition of the local maxima, two nodes in C_i cannot have an edge between them. Since Type 2 edge detection only involves Type 2 edges and edge e can never be a Type 2 edge, e does not affect the detection of Type 2 edges during step i. Finally, the edge e cannot be of Type 3, since a Type 3 edge must be between two undetermined nodes. Since c is in C_i this is not possible. Furthermore, Type 3 edge detection involves only Type 2 and Type 3 edges. It follows that edge e cannot affect the detection of a Type 3 edge during step i. Therefore, edge e cannot affect any other edge detection during pass i. Note that it obviously cannot affect any edge

detection in any step $j > i$ for the simple reason that $c \in C_i$ is removed from the graph and therefore so is e. Type 1 edges can therfore be removed from E_i immediately after detection or at any time convenient for the particular implementation.

Type 2 edges: An edge $e = (n, c)$ where $c \in C_i$ is a Type 2 edge, e.g., edges (v, c_1), (u, c_2), and (w, c_2) in Figure 2.2. Clearly, a Type 2 edge can never be a Type 1 edge and cannot affect the detection of Type 1 edges. Edge e also cannot affect the detecton of other Type 2 edges. Edge e may, however, affect the detection of a Type 3 edge. Consider edge $e = (u, c_2)$ in Figure 2.2, where $c_2 \in C_i$. By the definition of a Type 3 edge, edge (u, c_2) is required to detect the presence of the Type 3 edge (u, w). Therefore, if edge e is detected and removed from E_i before testing (u, w) for classification as a Type 3 edge, (u, w) would be erroneously not classified as Type 3. Since type detection occurs in no particular order, e must remain in the graph until at least the end of step i. As with Type 1 edges, since $c \in C_i$ is removed from the graph, it is not possible for Type 2 edges to affect the detection of edges in any step $j > i$. Type 2 edges, therefore, cannot be removed immediately upon detection but may be removed at the end of step i, or at any point where it is known that the edge cannot affect any other Type 3 edge detection.

Type 3 edges: An edge $e = (n, v)$ where (v, c) and (n, c) exist and $c \in C_i$ is a Type 3 edge, e.g., edge (u, w) in Figure 2.2. By the definition, the nodes of the Type 3 edge, n and v, must be the sources of Type 2 edges. Since the destination of a Type 2 edge is coarse and by the definition of the local maxima nodes n and v cannot be coarse. Thus edge e cannot be Type 1 or Type 2 based on C_i during step i and it cannot affect the detection of Type 1 or Type 2 edges during step i.

There are, however, three possibilities for an edge of Type 3 that must be considered. The edge $e = (n, v)$ could become a Type 1 edge in step $j > i$, i.e., $n \in C_j$, or a Type 2 edge, i.e., $v \in C_j$, or either n or v (or both) could be classified as fine. The first case, where e becomes of Type 1 when $n \in C_j$, does not require e to be kept in the graph beyond step i since Type 1 edges do not affect any other type of edge detection. So either e could be removed from the graph at the end of i or it could be kept until it is removed as a Type 1. Care must only be taken when considering the update of the weight v. Since it would be decremented when e is identified as a Type 3 in step i, its transformation to a Type 1

must be recognized in j and an additional erroneous update of the weight supressed. This is important because node v remains in the graph.

If the Type 3 edge e becomes a Type 2 edge when $v \in C_j$ then it may be involved in the detection of other Type 3 edges during step $j > i$. Since Type 2 edges are required only until the end of the step on which they are detected, in order to detect Type 3 edges the edge e can be removed at the end of step j. Note that since v is also removed from the graph at the end of step j there is no concern about an erroneous update of the weight of v. This transformation can be seen in Figure 2.2. If the Type 3 edge (u, w) is detected in step i when $c_2 \in C_i$ and if $w \in C_j$ for $j > i$, the presence of the edge (u, w) is required in step j to detect the Type 3 edge (u, z). Once the edge (u, w) fulfills its role as a Type 2 edge, it can be removed from the graph after step j.

Therefore, since it is not known know whether a Type 3 edge becomes a Type 1 or Type 2 edge later it can be marked for removal from the graph, i.e., included in the set Q in the form of the algorithm in Figure 2.1 and not removed from that set until it transforms into a Type 1 or Type 2.

Finally, either the source or the destination, n or v, could eventually become fine. In this case the edge (n, v) is no longer needed to drive any further removal. Consider the case where the source and destination of the type 3 edge are fine. If the destination of the edge v is a fine node, then it cannot be selected as a coarse point in some future C_j therefore, this edge cannot become a Type 2 edge, and cannot be needed to detect another Type 3 edge.

If the source of the edge n is fine, then all edges ending at n have been removed from the graph. However, since e is an outgoing edge and v may become coarse and transform e into a Type 2 edge that can influence Type 3 detection care must be taken. The easiest approach is to alter the definition of fine nodes to only finally declare them as such when all edges are removed. This is discussed further later. The fact that all incoming edges are removed prevents them from ever becoming coarse and influencing the results of CLJP.

In summary, Type 1 edges can be removed immediately, however, because these edges do not affect any edge detection but themselves, and they are never detected as a Type 1 edge twice within the same pass, it is safe for these edges to remain in the graph until the end of the step when Type 2 edges are removed. This may be beneficial to certain implementations of the CLJP algorithm. Type 2 edges must remain in the graph until the end of the step

during which they are detected (with a special check whether the edge has already been detected as a Type 3 to avoid erroneous weight updates). Finally, Type 3 edges may change to Type 2 edges on a later step and be needed to detect other Type 3 edges. Therefore, a Type 3 edge must be kept available, but not allowed to drive weight updates, until it resolves as a Type 1 or 2 edge or until one of its defining nodes become fine. This is done in the CLJP algorithm of Figure 2.1 by placing such an edge in the set Q.

CHAPTER 3

THREE BASIC VERSIONS

This chapter presents the three basic approaches to the CLJP algorithm that are modified and analyzed during the investigation. The abstract form of CLJP given in Figure 2.1 made no assumption about the manner in which the various edges were accessed, e.g., ordering, or represented, e.g., tuples, lists. Sections 3.1, 3.2 and 3.3 present less abstract forms by making a high-level assumption about the manner in which the graph is represented, e.g., the type of information that is easily accessed for a given node and by imposing a particular choice of the manner in which the graph is scanned, e.g., two passes through all nodes. The later sections discuss more specific implementation issues such as choice of data structure and synchronization implications.

3.1 Row-oriented version

This section describes the row-oriented version of the CLJP algorithm. The edge information is stored in terms of outgoing edges, i.e. the destination nodes of edges leaving node n are in S_n and easily available. Once the weights have been assigned to all nodes in the graph G_i and the local maxima has been constructed, the row-oriented algorithm comprises two passes each involving a loop over the nodes of G_i.

The first pass detects the three types of edges and marks them accordingly based on the set of coarse points C_i. The first pass of the algorithm is described in Figure 3.1. The first pass is organized by visiting each node of G_i. No assumption is made in this version of the order in which the nodes are visited. Each node is processed according to its status. The coarse nodes of graph G_i are processed by loop 1.1 that marks all of Type 1 edges by marking the destination node in S_n. Note that only edges specified by unmarked nodes in S_n are considered in the algorithm given. Any marked node in S_n at this point in the algorithm must be a Type 3 edge that is detected during a previous step. As discussed earlier, these

edges can be marked as Type 1 and the weight update suppressed. As is seen later, however, they can, in fact, be left in S_n when using the pass 1 algorithm in Figure 3.1 since they do not affect correctness only efficiency. The non-coarse nodes are processed by loops 1.2 and 1.3, this processing is more complicated than loop 1.1. Loop 1.2 detects Type 2 edges by marking all the nodes in S_n that are in C_i, i.e. the intersection of S_n and C_i. As noted with loop 1.1, processing only unmarked nodes ignores Type 3 nodes that became Type 2. Loops 1.1 and 1.2 add edges to R_i, the set of edges to be removed at the end of the pass. The algorithm, however, does not need to maintain explicitly a set R_i, instead only membership in the set for a given edge is maintained by marking that edge as Type 1 or 2 in S_n.

Once loops 1.1 and 1.2 have processed the edges in S_n, loop 1.3 detects edges of Type 3, those connecting two non-coarse nodes that depend on a common coarse node, and marks them in S_n. Given an edge (n, v) this is accomplished by checking to see if there exists a coarse node c, such that the edges (n, c) and (v, c) exist. Note this exploits the intersection $S_n \cap C_i$ computed in loop 1.2. If these edges are present then the edge (n, v) is of Type 3. This part of the algorithm is equivalent to the definition and update of the set Q in Figure 2.1. Like the set R_i this algorithm does not specifically maintain the set Q, instead edges of Type 3 are marked in a way that distinguishes them from the edges of Type 1 and 2 and unmarked edges.

A second pass through G_i identifies and removes C_i and F_i, as well as the examining the weights to determine new local maxima, C_{i+1}. After the first pass the weights of nodes in G_i have been updated, but in order to use a simple row-oriented pass through G_i to identify the local maxima some care must be taken. The implementation must be designed such that it avoids the need for an extra initialization pass through the vector that is used to store the status of the node, (fine, coarse, undetermined). The methods used to prevent the extra pass, are discussed later in Section 3.4. At this point, it is only important to note that the code assumes that a node currently in G_i is in C_{i+1} and the tests given in Figure 3.2 mark it (possibly multiple times) as non-coarse if it is not coarse. Note that in loop 1.6 only unmarked nodes are considered. This is necessary since some may correspond to Type 3 edges that have already updated the appropriate weights and therefore have no role in determining the local maxima neighborhood for C_{i+1}.

```
for each node n ∈ G_i
        1. if n ∈ C_i then (the node is coarse)
                1.1 for each node v ∈ S_n including Type 3 edges
                        1.1.1. if v ∈ S_n is not a Type 3 edge
                                1.1.1.1. decrement w(v) by 1
                        1.1.1. end if
                        1.1.2. mark node v in S_n as a Type 1 edge
                1.1 end for
        else (the node is undetermined)
                1.2 for each node v ∈ S_n ∩ C_i including Type 3 edges
                        1.2.1 mark node v in S_n as a Type 2 edge
                1.2 end for
                1.3 for each node v ∈ S_n that has no type
                        1.3.2 if (S_n ∩ C_i) ∩ (S_v ∩ C_i) ≠ ∅ then
                                1.3.2.1 mark node v in S_n as a Type 3 edge
                                1.3.2.2 decrement w(v) by 1
                        1.3.2 end if
                1.3 end for
        1. end if
end for
```

Figure 3.1. Row-oriented Pass 1 based on similar algorithm in [4]

There is, however, one complication that arises when using a row-oriented data structure. This complication involves the handling of fine nodes, and their removal from G_i. Since a node becomes fine as soon as its weight becomes less than 1, i.e. its column count is 0, it is possible to have a fine point that still has outgoing edges from the node. The difficulty arises not with the fine node itself but with the nodes to which it points via the remaining edges in the row. These destination nodes must become coarse during a subsequent step of the algorithm. If, however, the fine node is removed from G_i and the weight of the destination decremented, the destination may become a fine node, possibly violating the constraints of the coarsening process. As a result, in the row-oriented version, a node is made fine, and removed from G_i, when its weight drops below 1 and all of its row edges have been removed.

In addition to the removal of fine nodes, this pass is also used to remove the coarse nodes C_i from G_{i+1}, and the edges of Type 1 and 2 that are detected during pass 1 step i. Later

for each node $n \in G_i$
1. if $n \in C_i$ then (the node is coarse)
1.1 remove n from G_i
else if $w(n) < 1$ & $\|S_n\| = 0$ then (the node is fine)
1.3 add n to F_i and remove it from G_i
else (node stays for G_{i+1})
1.5 remove all edges of type 1 and 2 in S_n
1.6 for unmarked $v \in S_n$
1.6.1 if $w(n) > w(v)$ then $v \notin C_{i+1}$
1.6.2 if $w(n) < w(v)$ then $n \notin C_{i+1}$
1.6 end for
1. end if
end for

Figure 3.2. Row-oriented Pass 2 based on similar algorithm in [4]

in the discussion of other implementations, it is shown that nodes in C_i can also be removed from G_{i+1} in the first pass without difficulty.

3.2 Column-oriented version

It is also possible to perform the CLJP coarsening using a column-oriented approach. The edge information is stored in terms of incoming edges, i.e. the source nodes of edges ending at node n are stored in S_n^T. The weights of the nodes are measured by the number of incoming edges to a node, therefore this version is potentially much simpler than the CSR version concerning weight processing. When processing a node n all edge information which affects the weight of n is stored in S_n^T. This simplicity promises to yield a more efficient parallel implementation.

Like the row-oriented version, this algorithm updates G_i using C_i, by passing through all nodes in G_i. Nodes in G_i may be either coarse or undetermined and this distinction is used to define the column-oriented processing.

In loop 1.1 the node n is not coarse, all edges (v, n) where v is a coarse point are Type 1 and are removed immediately.

for each node $n \in G_i$
1 if $n \notin C_i$ then (the node is undetermined)
1.1 for each node $v \in S_n^T \cap C_i$ including Type 3 edges
1.1.1. if $v \in S_n^T$ is not a Type 3 edge
1.1.1. decrement $w(n)$ by 1
1.1.1. end if
1.1.1 remove v from S_n^T (**Type 1** edge)
1.1 end for
else (node is coarse)
1.2 for each node $v \in S_n^T$ including Type 3 edges
1.2.1 for each $p \in S = (S_n^T \cap S_v^T)$
1.2.1.1 mark node p in S_v^T as a **Type 3** edge
1.2.1 end for
1.2.2 decrement $w(v)$ by $\|S\|$
1.2 end for
1.3 remove n and S_n^T from G_i (**Type 2** edges)
1 end if
end for

Figure 3.3. Column-oriented Pass 1 based on similar algorithm in [4]

The processing of coarse-points makes up the majority of edge removal in this approach. In this case, the goal is to remove edges that point to the coarse node (Type 2), and identify edges between nodes that both depend on the coarse node (Type 3). This can be done quite simply in terms of the column structures, and the Type 2 edges can be removed upon completion of processing the coarse node, n, and not after the current step, as in the row-oriented version.

Essentially, for all coarse nodes n, each node, v, that is in S_n^T must be considered. Since it specifies an edge (v, n) where n is coarse all such edges must be removed (Type 2). Whether or not they are actually removed is an implementation question since the column and node n are never used by the algorithm again. Also, note that given a coarse node n, by the definition of the local maxima, all nodes in S_n^T must be non-coarse.

Because the removal of Type 2 edges happens when their coarse node is removed from the graph, the main goal of the processing coarse nodes is to identify Type 3 edges, such

as (u, w) in Figure 2.2 and to update, not S_c^T, but the S_w^T where $w \in S_c^T$. In the case of Figure 2.2, when processing S_{c2}^T, the information in S_w^T must be examined and updated since $w \in S_{c2}^T$. The edge (u, w) is a Type 3 edge and therefore u in S_w^T must be marked as Type 3. Since $u \in S_{c2}^T$, the node is easily identified as a member of $S_{c2}^T \cap S_w^T$ and all nodes in that set must marked as Type 3 in S_w^T (in this case only u).

for each node $n \in G_i - C_i$
1. if $w(n) < 1$ then
    1.1 add n to F_i
    1.2 remove n from G_i
1. end if
2. for $v \in S_n^T$
    2.1 if $w(n) < w(v)$ then $n \notin C_{i+1}$ end if
    2.2 if $w(n) > w(v) > 1$ then $v \notin C_{i+1}$ end if
2. end for

end for

Figure 3.4. Column-oriented Pass 2 based on similar algorithm in [4]

The second pass in this algorithm, shown in Figure 3.4 is basically the same as the second pass in the row-oriented version with the appropriate modifications. In this case, the coarse points are removed in the first pass. The computation of local maxima must be altered to use the primarily column information while preserving the bidirectional use of edges when determining the neighborhood over which the maximum is checked. As before, it is assumed that all remaining nodes are local maxima and if it is found otherwise, the node is marked as not in C_{i+1}.

Fine nodes with outgoing edges must be considered in a way different from the row-oriented version. When a node, f, becomes fine while an edge (f, v) still exists, care must be taken. The edge is stored as $f \in S_v^T$ and the weight of v keeps its contribution. Eventually v becomes a local maxima and v is made coarse or the edge from f to v becomes Type 3 and v becomes fine. The only concern, therefore, is when setting the local maxima flag of f while processing v the code must check if the source of the incoming edge is fine and therefore

avoid resetting its status to undetermined. Determining the status of f is accomplished in an implementation-specific fashion.

3.3 A Hybrid Version

The column-oriented algorithm has as its main processing step the update of undetermined nodes that are within the column structure of a coarse node. In that version, for each coarse node all of the columns of the undetermined nodes are updated based on the column of the current coarse node. While this form is appealing due to its simplicity, it has some potential problems with shared and distributed memory parallelism due to synchronization and communication.

It is possible to reorganize the column-oriented CLJP algorithm to make use of not only the column-oriented structures, but also the row-oriented ones (see Figure 3.5). In the hybrid algorithm, the detection of Type 1 edges remains the same as in the column-oriented algorithm. The difference comes from the manner in which Type 3 edges are detected. While processing a given node n, the hybrid version uses the S_n structure in combination with the S_n^T structure to detect all Type 3 edges ending at node n. This reorganization of the algorithm allows the same edge detection to be accomplished while only updating the weight of the current node n.

In order to understand the reorganization, consider node w in Figure 3.6. The edge from c_1 to w is contained in S_w^T and all such Type 1 edges are removed by loop 1 of pass 1 of the algorithm. The value of $w(w)$ must also be updated accordingly, this can be done after loop 1 by decrementing $w(w)$ by the number of nodes in $S_w^T \cap C_i$ (details of how this is handled varies between implementations). The edges of Type 2, from w to c_2 and u to c_2, are removed when all edges in $S_{c_2}^T$ are removed by the removal of the coarse node c_2 from the graph. Note that this does not affect $w(w)$ so no weight update is required. When seeking Type 3, however, the S_n structure must be used. This structure is used to detect the edges of Type 2 that contribute to the detection of Type 3 edges. Once the edges of Type 2 have been detected, S_n^T is used to find edges of Type 3. Edge (u, w) in Figure 3.6 is representative of these types of edges. Loop 2 considers edges like (u, w) by visiting each coarse node $c \in S_n$

for each node $n \in G_i - C_i$

1. for each $v \in S_n^T \cap C_i$ including Type 3 edge
    1.1. if $v \in S_n^T$ is not a Type 3 edge
        1.1.1. decrement $w(n)$ by 1
    1.1. end if
    1.2 mark node v in S_n^T as a **Type 1** edge
1. end for
2. for each $c \in S_n \cap C_i$ including Type 3 edges
    2.1 mark node c in S_n as a **Type 2** edge
    2.2 for each $x \in S_n^T \cap S_c^T$
        2.2.1 mark node x in S_n^T as a **Type 3** edge
    2.2 end for
    2.3 decrement $w(n)$ by $\|S_n^T \cap S_c^T\|$
2. end for

Figure 3.5. Hybrid version Pass 1 based on similar algorithm in [4]

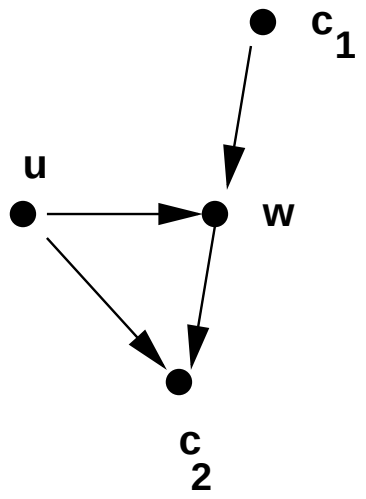

Figure 3.6. Hybrid updates

that is the destination of an edge leaving w. Then every node that is the source of an edge to c and w is marked in S_w^T as a Type 3 edge. Finally $w(w)$ is decremented by $\|S_w^T \cap S_c^T\|$.

If S_w is available then its updates are done at the same time as those of S_w^T. Keeping the column and row data structures consistent is only important relative to the coarse nodes that are selected on each step. In the algorithm above, note that Type 1 edges are marked in the S^T structure while Type 2 edges are marked in the S structure. This does not cause a problem with the coarsening because there is no interaction between Type 1 and Type 2 edges. While having these extra edges in each data structure does not effect the correctness of the coarsening, it can lead to extra processing of edges. Thus increasing the total execution time of the algorithm. For this reason it is worthwhile to explore the updates to the data structures and their effects on the execution time of the hybrid algorithm. This is investigation, however, is not included in this thesis, and is left to future work.

From the algorithm above it is easily seen that the only weight that is affected in processing a node is the weight of n itself. When implementing a parallel version of the algorithm this greatly reduces the amount of synchronization required. This topic is discussed further in Section 4.4

for each node $n \in G_i$
1. if $n \in C_i$ then
    1.1 remove n from G_i
1. end if
2. if $w(n) < 1$ and $S_n = \emptyset$ then
    2.1 add n to F_i
    2.2 remove n from G_i
2. end if
3. for $v \in S_v^T$
    3.1 if $w(n) < w(v)$ then $n \notin C_{i+1}$
    3.2 if $w(n) > w(v) > 1$ then $v \notin C_{i+1}$)
3. end for

end for

Figure 3.7. Hybrid Pass 2 based on similar algorithm in [4]

The selection of a new C_i and removal of points in F_i from the graph G_i in the hybrid version, as seen in Figure 3.7, is basically the same as that seen in the column-oriented version. Nodes with weights less than 1 are removed, and because of this it must be guaranteed the algorithm does not reset a fine node to an undetermined one. The one difference between the two versions is that in the hybrid version the coarse nodes are removed from G_i during the second pass rather than the first pass of step i.

3.4 Implementation

There are several ways to implement each of the versions above based on decisions concerning supporting data structures, parallelism, and architectural issues. In this section, preliminary discussions of several important implementation considerations are presented. These include: experimental framework, membership in G_i, edge storage, coarse-fine status, intersections, and synchronization. All of these considerations are revisited from the appropriate point of view in Chapter 4 when specific modifications to the basic implementations are evaluated.

3.4.1 Experiments

The basis for any multigrid coarsening process is the grid to be coarsened. The construction of the grid used for experimentation employs the following parameterization. For each grid, there are four parameters: dimension D, stencil P, boundary condition B and density K. All grids used in experimentation are square; therefore the dimension, measured in points, can be expressed as a single parameter, D. Given D, the number of points in the grid is defined as $D * D = N$. D can take on any range of values, a common value of D is 1,000, yielding an N of 1,000,000.

Once the size of the grid is established, the connectivity of this grid must be defined. The connectivity of the grid is defined by the three parameters P (stencil type), B (boundary conditions) and K (density). First, the basic connectivity of the grid is determined by applying a stencil to each point. This stencil, determined by P, can either be a nine or five point stencil. Figure 3.8 shows the result of applying either stencil to a single point in a grid where $D = 3$ (a 3×3 grid).

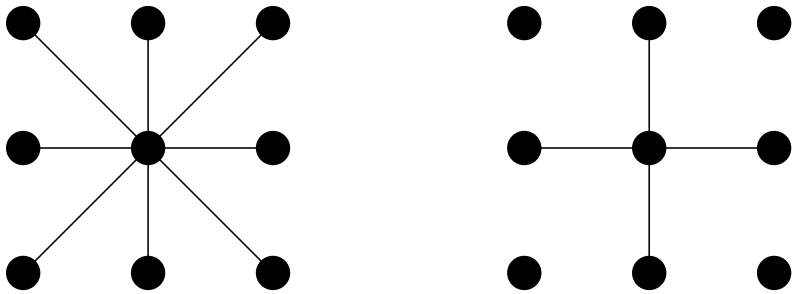

Figure 3.8. Example of a (a) nine-point and (b) five-point stencil

During the application of the chosen stencil to the grid, some points are on the edge of the grid. The way in which the construction of the grid handles these edge points is defined by B, the boundary condition. B differentiates between whether or not periodic boundary conditions are used. In the case where periodic boundary conditions are used, the stencil essentially wraps around the edge of the grid and connects to the other side. When periodic boundary conditions are not used, it can be considered that any points outside the grid do not exist, therefore no connections that leave the grid exist.

In addition to B, the grid density, K, also modifies the way the stencil, P, is used to construct connections. K specifies the maximum distance, in terms of the stencil type P, between two points that are connected. For example, a value of 2 for K means that the stencil is applied to a given point, n, creating connections between a number of neighboring points. Then the stencil is applied again to every one of those points and each point in these second stencils is connected to the original point, n. This essentially creates a connection between n and all points that could be reached in 2 steps on the normal graph (where $K = 1$). The size of the connectivity neighborhood (the number of points to which a point is connected plus itself), for a nine-point stencil, can be found based on K using equation 3.1

$$(2 * K + 1)^2 \tag{3.1}$$

Using the above process, a connectivity pattern on the grid is defined. For use in the coarsening algorithm this grid must be converted to a graph. This conversion process is simple, and never is actually performed in the code (the actual code constructs the graph directly). The conversion of the grid to a graph follows two steps: first for every point in

the grid a node is added to the new graph, then for every connection between two points in the previous grid, two directional edges (one going each way) are added between the corresponding nodes in the graph. Once this is complete, the local maxima is determined, and the algorithm proceeds to coarsen the graph.

The measurement taken for most experiments is total execution time, not including initialization of the graph. Results are often presented in terms of speedup between two or more versions of the code. This value is calculated by taking the execution time of the original code, $T(p, orig)$ where p is the number of processors, and dividing it by the execution time of the newly presented code, $T(p, new)$, i.e. $S(p) = \frac{T(p,orig)}{T(p,new)}$. The function $S(p)$ is then displayed on a graph. If $S(p)$ is greater than 1 the new code executes faster than the old code, in fact $S(p)$ is how many times faster the new code is compared to the old code.

In addition to the number of processors, K is also varied for most experiments. As multigrid coarsening proceeds the coarsening process is recursively executed to define a hierarchy of coarse grids. Each of these coarse grids is progressively denser. By varying K, from 1 to 6, this behavior is simulated with only having to perform a single coarsening.

Another common theme throughout the experiments is the value of D. In most experiments a D of 1000 is used. This value of D is chosen as it represents a typical and sufficiently large graph. Experiments were run for larger and smaller graphs, with the same results as the chosen graph. Therefore, the experiments presented use the typical graph.

All experiments are compiled and run on the same Sun Enterprise E4500 server, with 11 processors. Each processor has a 64KB level 1 cache and 2MB of level 2 cache. The total amount of shared memory is 8GB.

The code for the experiments is compiled using the "Sun WorkShop 6 update 2 C 5.3 (2001/05/15)" cc compiler. The flags -xopenmp and -fast are supplied to the compiler. -xopenmp turns on OpenMP support, while -fast sets a number of performance options intended to yield maximum performance, including -x05 (optimzation level 5), and -xtarget=native (compile for this machine's ISA only). The sunmath library is required for compilation (command line option -lsunmath), however, the function call that requires sunmath is part of the problem setup procedure (which is not timed) and could be easily replaced for portability reasons. More specifically the function used is $d\_mwcrans\_()$, which fills the $w()$ array with pseudo random numbers, as required by the CLJP algorithm. This

function could easily be replaced by a loop and a call to any other pseudo random number generator.

3.4.2 Membership in G_i

In all versions mentioned in the previous section, membership of a node in G_i, is maintained. This membership must allow easy association of the nodes in G_i with a loop schedule that may or may not be parallel. The easiest way to accomplish this is by using an indirection vector, $graph\_index(1:N)$, that is initialized to 1 : N. A loop over all nodes in G_i is then simply a loop from the position after the removed region to N. Removal from G_i corresponds to moving an index of a node from the region of the vector that contains nodes in G_i to one that contains those that have been marked coarse or fine. To allow for easier removal, assuming sequential processing, this region of coarse and fine nodes is located at the beginning of $graph\_index$. As long as no ordering is assumed on the nodes in G_i, node removal can be accomplished in $O(1)$ time by swapping the index to be removed with the first index in G_i and incrementing the initial position of the removed region by 1.

Swapping the nodes with the start rather than the end allows for an easier sequential implementation. It is known that if node $graph\_index(n+i)$, for some positive constant i, is being processed, then node $graph\_index(n)$ has already been processed. Therefore if a node $graph\_index(start+i)$ is being processed, $graph\_index(start)$ has already been processed. $graph\_index(start)$ can then be swapped with $graph\_index(start+i)$ and the algorithm can move on to processing $graph\_index(start+i+1)$. If the inactive region grew from the end of the list, after swapping, the current position in the active list would have to be processed again, which is not as simple from an implementation point of view.

During a sequential implementation the order in which the nodes are processed is guaranteed. In a parallel implementation, however, there is no guaranteed ordering, in general, to the processing of the nodes. In this case a new method of node removal must be developed. This new method is discussed in more depth in Section 4.1.

3.4.3 Edge Storage

For the three basic abstract versions presented in this chapter, edge data is stored in either the set S_n, the set S_n^T or both. These sets are simply a vector containing indices of the destination or source node in the graph with the second node implicit in the set name. If edge removal is not used, all edges can be represented by two states (1) active, denoted by i and (2) marked as removed, denoted by $-i$. Because these states are a subset of the states required for edge removal, their discussion is deferred until later in this section. When using edge removal there are four states in which the edges can exist: (1) active, (2) marked as Type 1 or 2, (3) marked as Type 3, and (4) removed. In most cases edges move from (1) to (2) to (4), where they remain. It is only in certain cases that edges must enter State 3, this is discussed at length in Section 2.3.2. Essentially, edges of Type $(g, v) \mid g, v \notin C_i$ must not be removed from the graph under the possibility that v could become coarse in a later pass, and the edge (g, v) would be required to drive the removal of an edge between two other non-coarse nodes.

Each state requires a unique representation in the edge data structure. The four chosen representations, for an edge with an index i, are as follows: (1) every edge starts out in State 1, (2) State 2 then can be represented by setting the edge index to $N + i$, (3) an edge is in State 3 if its index is $-i$, and (4) an edge is in State 4 if it is absent from the data structure.

Most edges start in State 1, proceed through State 2 to State 4. This process is accomplished by, first setting the edge index, i, to $N + i$. On a later pass, all edges in State 2 are removed via a swap method similar to the one used for the vector $graph\_index()$. When removing the edges it is known that all the edges in State 2 may be removed during a single pass through G_i, i.e., Type 1 and Type 2 edges can be removed together. Therefore, if a sorted vector is used to store edge information, there is the option of temporarily allowing the vector to become unsorted, only to sort it again when the edge removal is complete. Depending on the number of edge removals required, either choice could lead to better performance, both are explored further in Section 3.4.4. The remaining state, that of special edges (State 3), is indicated by negating the value of the edge index, i,. These edges remain in the graph for further consideration until it is decided that they are to be removed, by the same test that determines if active edges should be removed. When it is determined that

edges in State 3 can be removed, they then enter State 2, just like edges in State 1, and on the next pass are removed thus entering State 4.

Edges that are marked to be removed (Type 1 and 2) are not used in the determination of the local maxima edge removal, and therefore, edge removal can be performed during pass 2 of step i as each node in G_i contributes to the construction of C_{i+1}. When the edges are detected as Type 1 or Type 2 rather than ignore them, a simple removal process is performed. Because each node is visited once during pass 2 this process can be done in parallel without any synchronization.

3.4.4 Edge Removal

Removal from a sorted list can be done in one of three ways: (1) by re-sorting the list immediately after every edge removal is performed, (2) by waiting until all edges from a list are removed, then re-sort the list, or (3) by removing edges in such a way that does not disturb the ordering of the edge list. Due to a more efficient implementation, the latter of the three is used. This type of edge removal can be achieved using the following method.

The edge list is passed over from the front to back. Once an edge that needs to be removed is found, it is copied to the temporary storage. At the same time a shift value is incremented by 1. This shift value is the amount of distance each successive item in the list must shift, to be in the correct final position. At first, since only one edge has been removed, the successive edges must only shift by one. As more edges that can be removed are found, the shift value increases, causing the following edges to be shifted further (to compensate for the increased size of the gap between active and removed edges). Once the entire list has been processed, the contents of the temporary storage are copied to the end of the list, inside the removed region.

3.4.5 Removal Effectiveness experimentation

Figures 3.9-3.12 show the results from the removal effectiveness experimentation for the three versions. In these experiments both node and edge visits are counted. Note that node removal does not affect the number of edge visits, and edge removal does not affect the number of node visits. A node visit is defined as a single iteration of the pass 1 outer loop

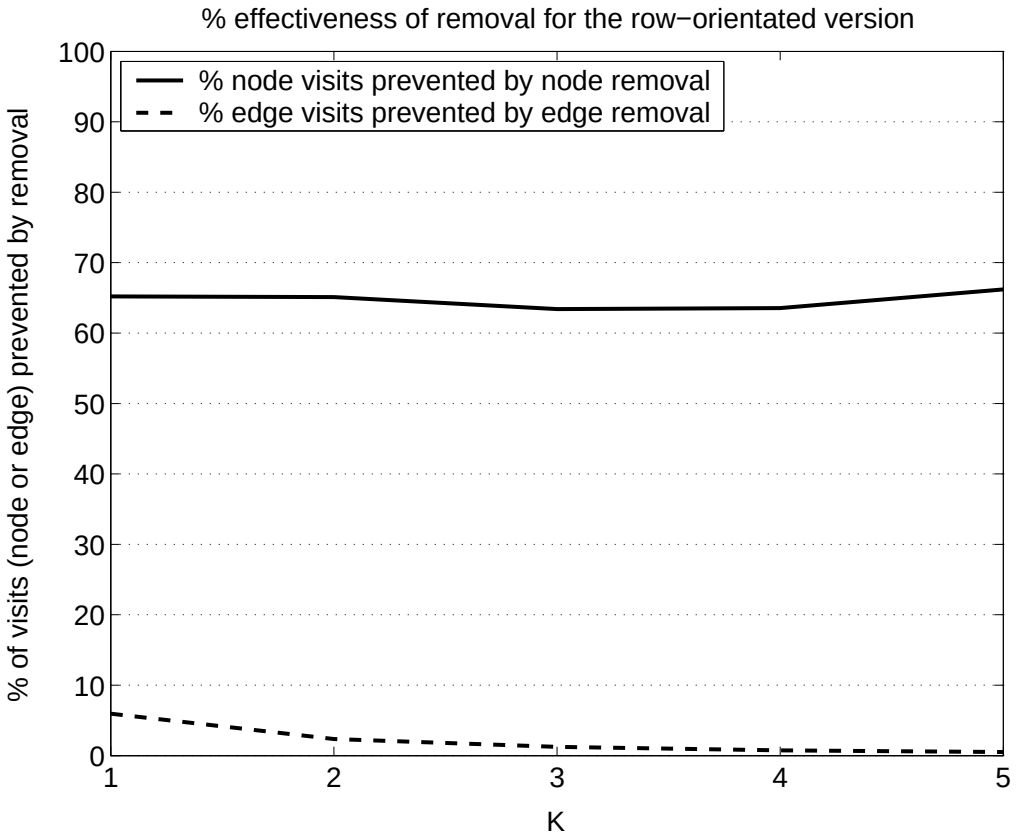

Figure 3.9. Effectiveness of node and edge removal for the row-oriented implementation.

over all nodes in the graph, or in the case of the hybrid, all undetermined nodes in the graph. An edge visit is similarly defined as a single iteration of an inner loop over all nodes in a certain edge list (S_n or S_n^T). To clarify this definition, outer and inner loops are defined. In Figure 3.13 the outer and inner loops of the basic row-oriented algorithm are labeled. This definition of outer and inner loops is important since it is also used in the discussion of outer vs. inner loop parallelism in Chapter 4.

In these experiments the number of node and edge visits in a complete coarsening process of a one million node grid ($D = 1000$) are recorded for several density levels, $K = 1 : 5$ with and without node removal, edge removal, and both removals. The figures show the percent reduction in the number of node or edge visits caused by node and edge removal, respectively. This experiment makes a strong case for node removal showing an average of 75% reduction

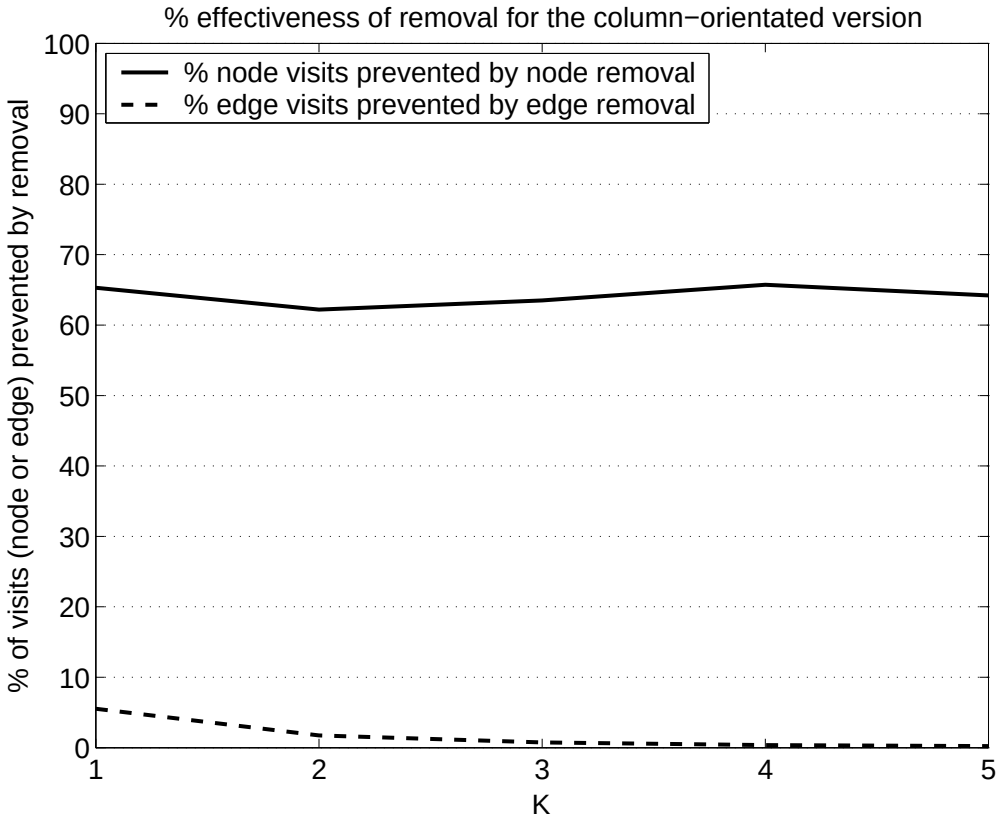

Figure 3.10. Effectiveness of node and edge removal for the column-oriented implementation.

in nodes visits for all levels of density. However, the experiments in Section 4.1.4 show that this reduction in work, in fact, does not lead to a reduction in execution time.

Edge removal, however, is shown to have little affect on the number of edges visited. Across all versions less than 5% of the total edge visits are prevented by enabling edge removal. In order to understand why this is the case, consider what happens when there is no actual edge removal, only edge pruning. When there is no edge removal all edges can be in one of two states, mentioned in Section 3.4.3. These two states are, (1) active, denoted by i, and (2) marked as removed, denoted by $-i$. In this case, marked as removed is defined as all edges that have previously driven a weight update (this includes Type 3 edges). Using these two states, and a simple test of i, edges that have been marked as removed can be excluded from processing; this method is called edge pruning. The result is that the difference, in

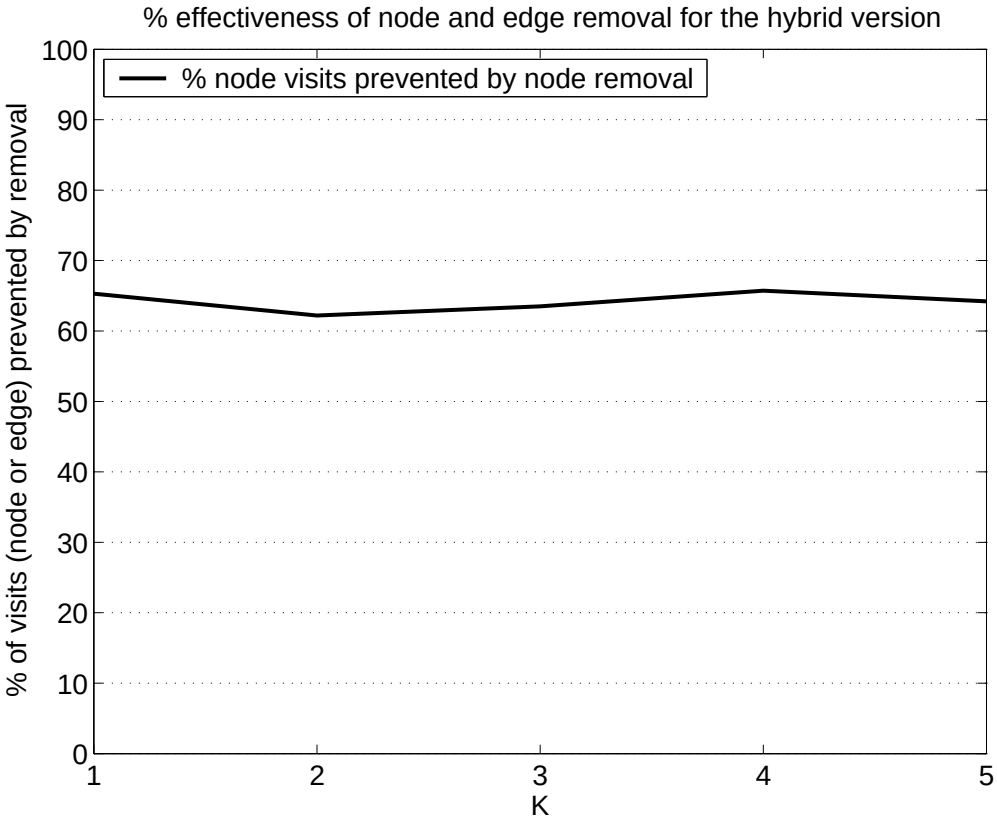

Figure 3.11. Effectiveness of node and edge removal for the hybrid implementation.

terms of edge visits, between real edge removal and edge pruning is marginal ($< 5\%$). The lower overhead of the simpler pruning method leads to better overall performance, as seen in Figure 3.14, than the original edge removal implementation.

The experiment in Figure 3.14 is performed using a standard $D = 1000$ graph with a nine point stencil. K is varied from 1 to 3, and the experiment is run on 1 to 11 processors. $T(p, K, prune)$ is defined as the basic parallel implementation with edge pruning, while the actual edge removal is defined as $T(p, K, remove)$. The speed up $S(p, K) = \frac{T(p,K,prune)}{T(p,K,remove)}$ is presented in Figure 3.14. This experiment shows that when the number of processors is low the extra overhead of actual edge removal inflicts a greater loss in performance. As the number of processors increases, the total overhead cost lessens. However, due to the asymptotic nature of the curve, it is unlikely that edge removal is ever be faster than edge pruning, even on a large number of processors.

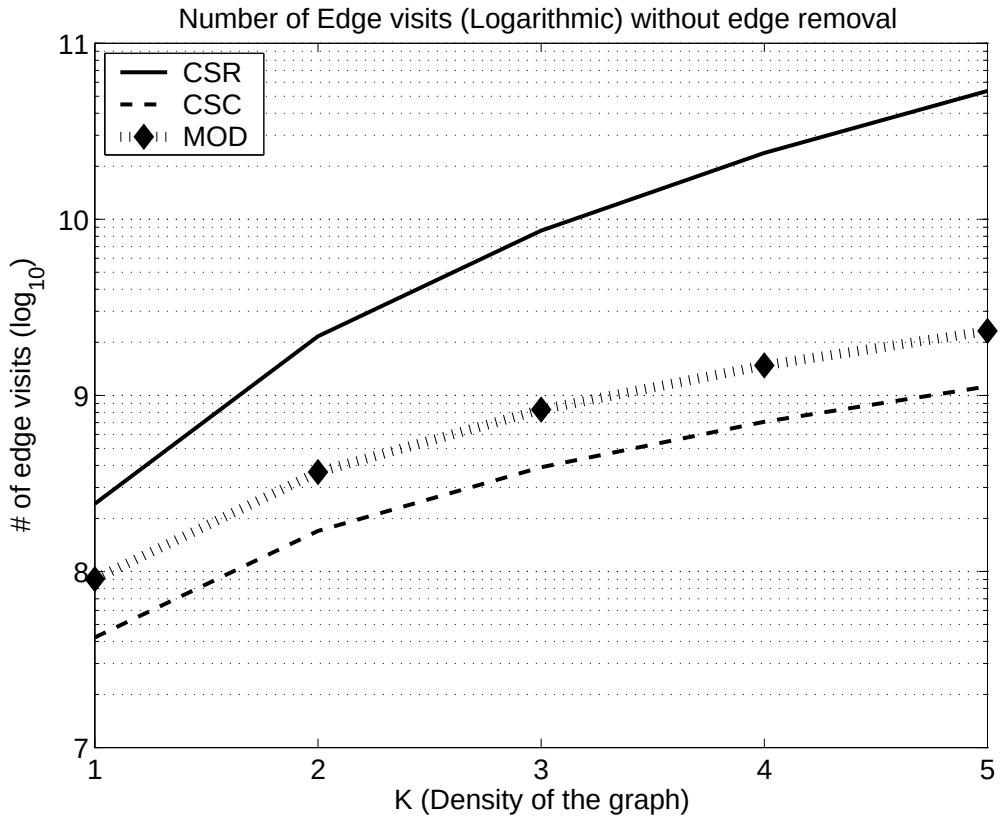

Figure 3.12. The number of edge visits for each version without edge removal (logarithmic scale). Same instrumented statistics used in the measure of effectiveness (Figures 3.9-3.11)

The performance difference between edge removal and edge pruning verifies that edge removal does not prevent a significant amount of additional work, when compared to edge pruning. The 5% additional edge visits do not account for enough work to make the complicated edge removal worthwhile.

3.4.6 Coarse-Fine Status

Coarse-fine status is represented by a special status vector, denoted the cf vector. The cf vector is used to indicate the status of a particular node in any of 3 states: (1) coarse, (2) fine, and (3) undetermined. Membership in each of the 3 states is indicated by the value of $cf(n)$ for node n. The corresponding values, for a given step i, are as follows: (1) if node is a member of the coarse set C_h, then $cf(n) = h \mid 0 < h \leq i$, (2) if the node is fine,

begin outer
for each node $n \in G_i$
1. if $n \in C_i$ then (the node is coarse)
begin inner
1.1. for each node $v \in S_n$ that has no type
...
1.1. end for
end inner
else (the node is undetermined)
begin inner
1.2. for each node $v \in S_n \cap C_i$ that has no type
...
1.2. end for
end inner
begin inner
1.3. for each node $v \in S_n$
...
1.3. end for
end inner
1. end if
end for
end outer

Figure 3.13. Abbreviated row-oriented pass 1. With outer and inner loops marked.

then $cf(n) = 0$, and (3) if the node is undetermined, then during step i, $cf(n) = i + 1$. The design of this structure is motivated by the aim to avoid an extra pass through the undetermined nodes of G_i, in order to reinitialize the values of cf that are updated during the determination of the next set of coarse nodes.

The extra pass is prevented by using the following method for determining coarse and fine nodes. At the start all nodes have values of 1 in cf, this indicates their presence in C_1, the first set of coarse nodes. The checks on the weights of the nodes are performed by going through each node of the graph, and determining which nodes should not be in the coarse set. The value in cf for these non-coarse nodes is incremented to 2, in preparation for the

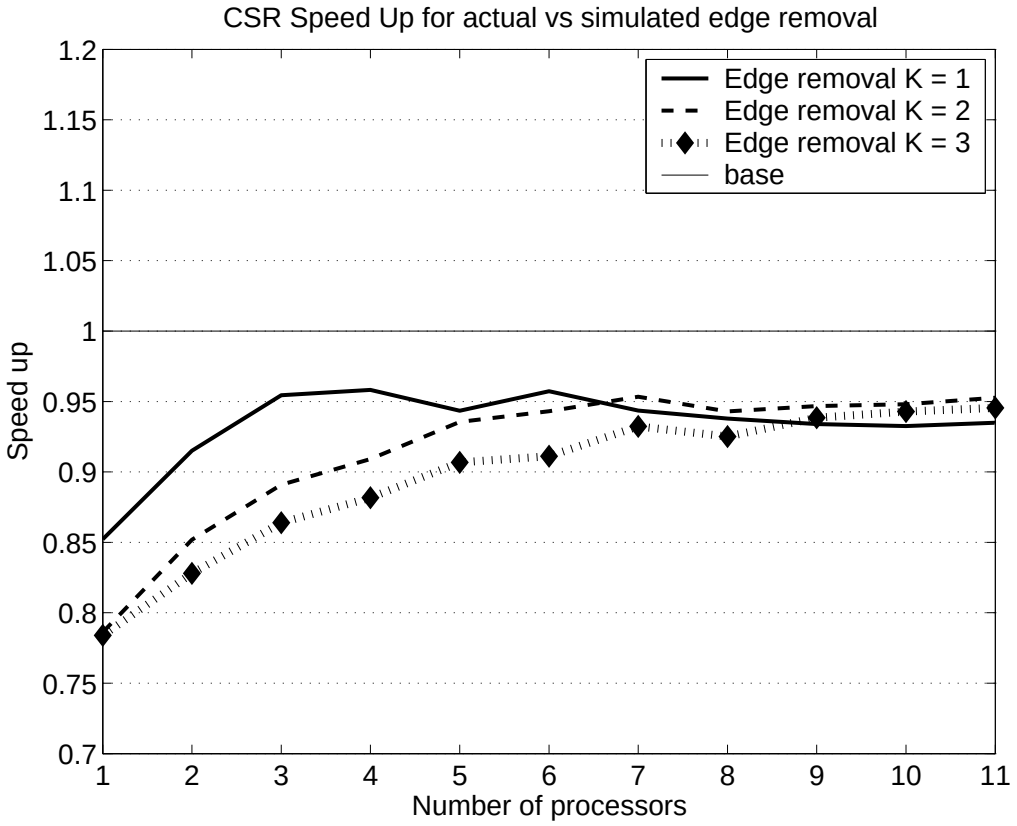

Figure 3.14. The speed up for edge removal vs edge pruning

next pass through cf when C_2 is needed, i.e., the earliest that the nodes can be made coarse is step 2.

If the non-coarse nodes are denoted by a 0, then another pass would be required at each level, to reinitialize the cf vector. Consider this example, say 0 represents a coarse node in C_i for pass i, -1 represents a fine node, and 1 represents an undetermined node. The whole cf vector can be initialized to 1 to start, representing that all nodes are undetermined until they are found to be coarse. The local maxima algorithm is applied to graph, a set C_i of coarse nodes is determined and denoted by 0s in the cf vector. The CLJP algorithm is applied, and then a new set of coarse nodes must be determined based on the resulting graph. Now because 0 denotes the old coarse nodes, these values must be changed in preparation for a determination of a new coarse set. A simple reinitialization strategy a pass over the entire cf, which is of size N. This can be improved by noting that only active nodes are of interest

in cf. The vector, $graph\_index$, used to store the indices of the active nodes could be used to reinitialize the subset of cf still of interest. By defining a sequence of coarse nodes with the value of i, the current pass, and defining undetermined nodes as $i+1$, the cf is automatically initialized on each pass of the algorithm and therefore no additional reinitialization activity is needed.

3.4.7 Intersections

In the sequential versions of the algorithm the vector cf can be used to identify intersections. In particular, the intersection of a set with C_i is easily determined by checking the value of $cf(v)$ where $v \in S_n$, if it is found that $cf(v) = i$ then it is know that $v \in C_i$. Additionally when processing non-coarse nodes later in the algorithm, the intersection of several sets, say: for each $v \in S_n$ find $S_n \cap S_v$, can be found for little work by performing the following. First the values of $cf(m)$ where m is a node in S_n are temporarily set to $-cf(m)$. The nodes that are in $S_n \cap S_v$ for several different v's can be found by checking $cf(w)$ for each $w \in S_v$. This method also applies directly to the column data structure, S_n^T.

Intersections need not always be found using information scattered into a vector of length N. If the indices in the row or column are sorted, an intersection is easily computed using one pass through the row or column pair. As seen earlier in this section, maintaining sorted order does complicate the removal process when an array-based data structure is used. A more in-depth analysis of implementation specific methods for sorted edge removal and computing intersections are discussed later in Chapter 4.

3.4.8 Synchronization

When parallelizing any algorithm, attention must be paid to the cost of performing synchronization between processors. With OpenMP there are two fundamental types of synchronization, lock-based and barrier. Several key differences exist between these two types, and with these differences come tradeoffs.

Lock-based synchronization Lock-based synchronization is used to protect critical blocks of code which modify objects in the shared data space. Before entering a critical

block of code, the processor sets a certain lock, the lock remains set until the processor unlocks it, usually when it is done updating the shared data. Any other processors that try to acquire the same lock, while it is set, are forced to wait until the lock is released. Lock-based synchronization is the simplest form of synchronization and tends to require less modification to the algorithm than other forms of synchronization.

Barrier synchronization Barrier synchronization involves a point in the code, usually after a loop, where all processors must complete the work above the barrier, namely the loop, before any processor can proceed past the barrier. In order to take advantage of barrier synchronization it is required that updates to the global shared data be delayed until after the current parallel loop. A common method of delaying the update to the shared data is to add processor local storage where updates can be accumulated before they are merged with the global data at the end of the loop. This, however, requires an additional parallel, or sequential loop to be added after the barrier. Chapter 4 contains more detailed specifics about various implementations that take advantage of barrier synchronization.

Experiments A simulation is used to estimate the amount of time required by each of the two possible synchronization methods. The model represents the cost of acquiring and releasing a single lock, when there is no possibility of contention for that lock. The model is run on a single processor/thread, using an OpenMP parallel-for-loop. The loop performs a series of non-trivial updates of a shared array, one of which is protected by a lock. The operations, while they serve no purpose, are designed to prevent the compiler from optimizing them away. For each type of synchronization two versions of the code are compiled and run; one with synchronization, and one without. The difference between the execution time of these two loops divided by the number of iterations (twenty million) yields the average cost per set/unset pair of lock operations.

For barrier synchronization the same loop is used, except instead of locks a barrier is added. After the barrier there is a second loop with a single update operation. This represents the pass over the temporary storage and the update of the global data structure required when using barrier synchronization. So that barrier synchronization could be compared with lock-based synchronization the difference between the two loops is once again

divided by the number of iterations. While the cost per loop iteration has no meaning for barrier synchronization, it still represents the average affect of barrier synchronization on the execution time of the loop.

The final measurements in Table 3.1 are the averages of ten runs. While critical sections are also included in the testing, they are not used or discussed in Chapter 4 since, in OpenMP, they can only be differentiated by symbolic compile-time constants. As a result, in order to use critical sections effectively in the CLJP algorithm the nodes must be partitioned into a number of groups that are independent of the number of processors used at run-time. Critical sections are included in this model to determine if the cost is low enough to warrant further inspection. Given the small difference in cost, especially compared to barrier synchronization, a critical section implementation is not investigated.

Type of sync	Time(ms)	% Slower	Cost per unit: (ns)	(cycles)
No sync	1070.117	Baseline	N/A	N/A
Critical	2342.285	118.88%	159	63
Locks	2492.141	132.88%	177	71
Barrier	1753.801	63.89%	85	34

Table 3.1. Times for various methods of synchronization (average of 10 runs)

From the results shown in Table 3.1 it is apparent that the cost of a lock is quite significant. Even without competition between processors the average cost per lock set is 177ns. Given the clock rate of the machine (400Mhz), that translates to approximately 71 cycles per set of locks. Barriers, on the other hand, cost 34 cycles when the overall cost of the barrier is divided by the number of loop iterations. Section 4.1 discusses in more detail the tradeoffs between locks and barriers, and various methods to employ a combination of the two to achieve maximum performance.

CHAPTER 4

MODIFICATIONS

4.1 Compressed Sparse Row

This section explores the various modifications that are made to the row-oriented version of the CLJP algorithm during implementation. For each version a number of defining features are discussed, including data structure choices and algorithmic changes. Arguments are given for the choices made, and experimental data is provided to support or reject these arguments. For all of the following versions, the cf vector is implemented as discussed earlier, therefore it is not mentioned. Any other aspects that are not mentioned can be assumed to be implemented in a way corresponding to previous descriptions.

4.1.1 Data structure definition

For the row-oriented algorithm the choice of data structure is simple. The desire to be compatible with BoomerAMG, and to have a simple and flexible data structure suggests a *compressed sparse row* (CSR) format[8]. The CSR format allows a matrix with with several zero elements, i.e. a sparse matrix, to be represented in a compressed format, thus saving memory and reducing access costs. The standard CSR format consists of three arrays. The first array AA contains all the non-zero elements of the original matrix, $a_{(i,j)}$. However, since the matrix is a connectivity matrix and all non-zero elements are 1, this array does not need to be represented in this implementation. The second array JA contains the column indices of the non-zero elements that are in AA. This array, JA, is similar to S mentioned before, the column indices of the connectivity matrix are the node ids of nodes with incoming edges, represented by a 1 in the connectivity matrix. The third, and final array IA consists of

pointers to the beginning of each row in both AA and JA. Thus, $IA(i)$ is the location in JA of the list of non-zeros of the ith row of a (the edge list).

For this implementation a few adaptations to this general definition of CSR are made. The JA array, which is now referred to as S, is exactly the same as defined. The chosen method for performing intersections of edge lists (each row of the original matrix), requires that the edge lists in S are sorted. Therefore, the column indices associated with a row are stored in increasing order in S. A slight modification is also made to the IA array. Instead of having a single IA array to point to the start of the edge lists, in S, two arrays of pointers S_ps and S_pe are created. These contain the starts and ends of each edge list in S. The second array of pointers is added to simplify the processing and removal of edges from the edge list.

The edge list is processed by starting at the front, and moving across each node in the list, until the end of the list is reached. Intersections are performed by simultaneously moving across two lists only processing nodes that are in both lists. This method of computing the intersection does require that the edge lists are sorted, this is why this implementation imposes an order requirement on the S array. If there is no actual edge removal, sorted order can be maintained throughout the coarsening process for free. During coarsening without edge removal, the only changes made to the edge list are the various edge type classifications, and the marking of an edge as removed. As described in Section 3.4, this is accomplished by negating the node id in the edge list which represents the edge that should not be considered. When edge removal is performed the method of maintaining a sorted list of edges explored in Section 3.4.3 is used.

4.1.2 Level of Parallelism

This section describes the various choices presented when parallelizing the algorithm. The first of these choices being, what loop to parallelize. Each pass of the CLJP algorithm consists of two types of loops, as defined in Section 3.4.5, the outer loop over the nodes n from 1 to N, and the inner loops over the edges of node n. Either of these loops can be parallelized with various tradeoffs between the two options.

Outer loop parallelism When the outer loop of each pass is parallelized the amount

of code inside the parallel section is much larger, greatly reducing the overhead to real work ratio. Including most of the algorithm inside of a single parallel block also increases the number of instructions that can be executed while in parallel, thus making the most efficient use of the available resources. Even though this method is the most efficient from the scheduling overhead point of view, it presents a few problems. When an edge is removed from the graph the weight of its destination node must be updated. In the CSR algorithm, due to the fan-out nature of coarse-driven edge removal, and the removal of Type 3 edges, it is possible that two different processors could remove two different edges whose destination is the same node. The weight information for each node is stored in the shared data structure, therefore the update of a weight must be protected by a lock. This locking behavior is the most significant source of non-work delay in the algorithm at this level of parallelism.

After weight updates, the second most important concern with outer loop parallelism is the processing of intersections. When working in parallel, using a scatter vector for the intersection would require a different vector of size N for each processor. The alternative to this excessive use of memory, is to require the edge list to be sorted. If edge pruning, which is faster than edge removal, is used, then no action is required to maintain sorted order. Due to the excessive size requirements of the first method for intersections and the simplicity of the second method, the latter is used.

Inner loop parallelism Many of the concerns about the parallelization of the outer loops can be avoided completely when working with inner loops. When the parallelism is over the inner loops, weight updates do not require locks. Since there is no parallelism across different nodes, two edges that point to the same node can not be removed by different processors at the same time.

Inner loop parallelism also allows for more efficient intersections, the use of the scatter vector in combination with inner loop parallelism can be beneficial. The nature of intersections performed in the CLJP algorithm are such that a single set can be scattered into a work vector, in parallel. That work vector can then be used to perform a number of intersections, also in parallel. As long the density of the graph is sufficiently high, i.e. the number of edges per node is sufficiently larger than the number of processors, this method can be efficient.

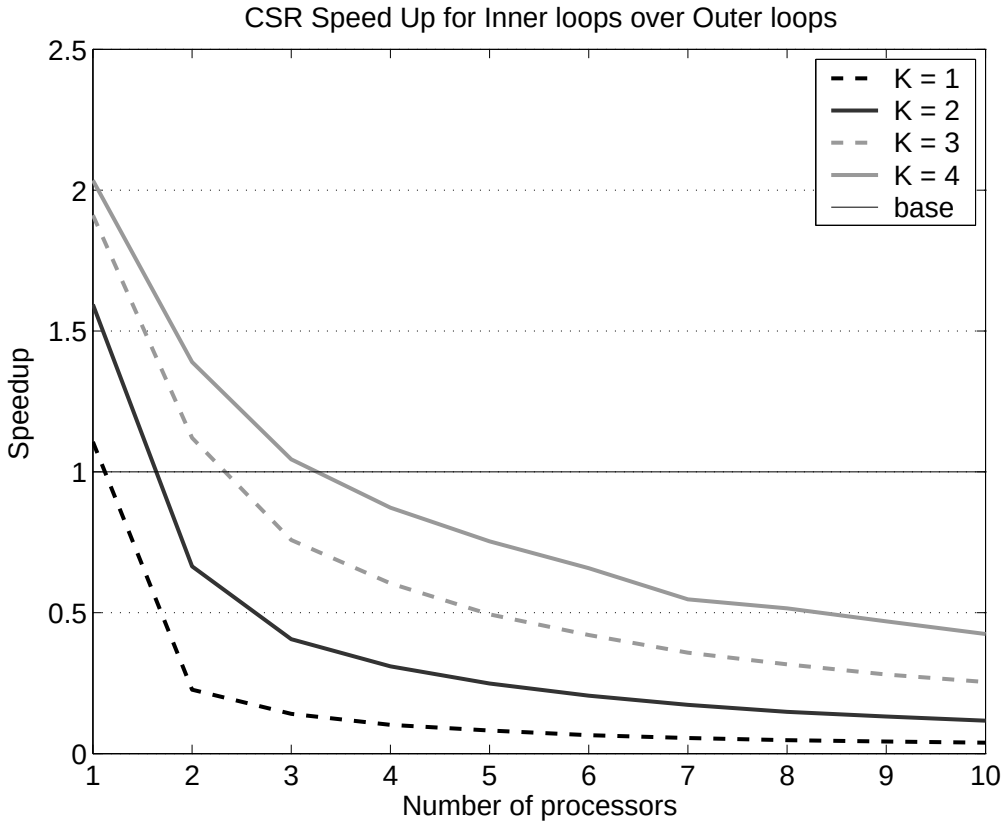

Figure 4.1. Speed up comparison for outer loop parallelism and inner loop parallelism

Comparison between levels of parallelism

This experiment is run on on the standard graph $D = 1000$, with $K = 1 : 4$, and on 1-10 processors. The synchronization calls are left in for all numbers of processors, including 1. For $T(p, orig)$ the outer loop, as defined above, is parallelized. For $T(p, new)$ the inner loop, also defined above, is parallelized. The speedup $S(p) = \frac{T(p,orig)}{T(p,new)}$ for various densities (K), is shown in Figure 4.1.

Because of the extremely small amount of work done in the inner loops, only the loop that detects edges of Type 3 could be parallelized in such a way that the execution time of the algorithm was reduced by parallelism. When detection of Type 1 or Type 2 edges is parallelized with inner loop parallelism the execution time increased from the sequential version. The loop overhead of processing across 10 processors is considerably larger than

the work done inside either of these loops. The significant overhead in processing a parallel loop causes the inner loop parallelism to be only beneficial for very dense graphs, and on a limited number of processors.

The graph in Figure 4.1 shows the speed up achieved when using inner loop parallelism instead of outer loop parallelism. At first, even for low density graphs there is a significant speedup when running the parallel code on 1 processor. There are two reasons for this speedup, the first reason is because weights can be updated without lock-based synchronization. As shown in previous experiments concerning the locking behavior of the Sun Enterprise E4500, even if there is no collision, as is the case on one processor, locking takes a significant number of cycles. When the work being done inside the lock takes only a few cycles, a penalty of 71 cycles affects performance considerably. The second reason for a speed up on 1 processor is because the intersections in the inner loop version are slightly faster than those performed in the outer loop version.

As the density of the graph increases, the speed up for 1 processor increases as well. When the density is increased the number of edges per node goes up, thus increasing the amount of weight updates that must be performed per node. The major performance increase in the inner loop version comes from the removal of locks on weight updates. Therefore, as the number of weight updates increases so does the performance.

This trend holds true as the number of processors increases. The most significant cost of inner loop parallelism is the loop overhead required to start processing on several processors. Though the reduction in the number of locks helps to offset this overhead, there just is not enough work done in the inner loop to make up for the overhead, even at high levels of density. As a result of this performance analysis, outer loop parallelism is used throughout all parallel implementations of the algorithm.

4.1.3 Parallelization

The use of outer loop parallelism, forces all weight updates to be protected by a lock. This is due to the fan-out nature of the CSR algorithm, and the method by which Type 3 edges are detected. This aspect of the algorithm sets up the possibility that two edge removals on different processors may try to update the same weight at the same time. To

prevent conflicting updates to the shared data structure, a lock must be acquired before a weight can be updated safely.

Intersections When using outer-loop parallelism care must be taken in choosing an efficient method of computing the intersection of two edge lists. By using an ordered edge list intersections can be performed in a single pass over both lists. Additionally, as long as there is no edge removal, no time is spent maintaining sorted order of the edge list. For the reasons mentioned above, this algorithm uses the sorted edge list method of intersection.

Lock granularity When using locks for synchronization, a key choice must be made involving the lock granularity, or ratio between the number of locks and the number nodes. Through analysis of the algorithm, it can be determined that different processors can be in the same block of code at the same time, as long as they are not updating the same shared data. Therefore, since only weight updates require protection from locks, the finest granularity that can be used is one lock for each weight, i.e. one lock per node. However, while this extreme measure reduces contention, it also hurts cache performance. By requiring each processor to acquire a lock for every node that it updates, the number of writes to memory per weight update is doubled (from 1 to 2). This increases false sharing, in that when one processor writes to a cache line, all other locks on that line are flushed from the other processors' cache, thus causing an increase in the total number of cache misses.

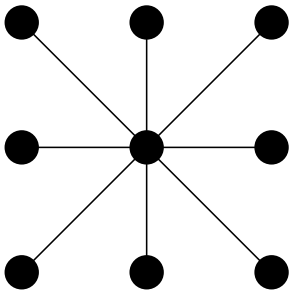

Figure 4.2. The bi-directional connections between a single node in a graph constructed with a nine point stencil and a value of 1 for K.

The effects of this fine lock granularity can, however, be mitigated by using a mapping of nodes to locks, where there are LG nodes per lock. This method of mapping more than one node onto a lock is meant to exploit the locality of weight updates found in the CSR algorithm when removing Type 1 edges. Consider Figure 4.2, if the center node is coarse then the edges leaving that node will be removed as Type 1 edges, thus causing the weight update of every node to which the center node is connected. Using the natural ordering the graph, it is known that each of the three rows of nodes will have consecutive node ids. Therefore, if these three nodes are mapped to a single lock, by a simple $node\_id/LG$ operation, then the update of these three nodes, while still requiring three separate lock operations, will use the same lock each time thus improving cache performance. Also, as the density, K, increases the rows of consecutive node ids will grow larger, allowing more nodes to be mapped to a single lock. It would be even more beneficial if a single lock operation could be used to update all of the weights that map to a certain lock, however, as the density of the graph increases and the stencil becomes more complex, the method for accomplishing this becomes increasingly difficult, and is expected to lead to an overall decrease in performance.

Lock granularity experiments This experiment is run on on the standard graph $D = 1000$ constructed using the standard nine-point stencil, with $K = 1 : 6$, and on 10 processors. The version tested is the basic parallel implementation described above, with a modification to the locks protecting the weight updates. The basic parallel implementation uses, by default a lock granularity of 1. In this implementation there is an array of locks of size N indexed by the node id whose weight is to be updated. When that lock is set, no other processor can update that weight. The modification made in this experiment defines an array of size N/LG, where LG is the lock granularity, which varies from 1 to 16 in variable increments. The array of locks is then indexed by a new mapping n/LG, where n is the node id of the weight to be updated. Because of the round off in the integer divide operation, this maps LG nodes to a single lock.

The execution time can be parameterized in terms of a time function $T(p = 10, K, LG)$. The speed up: $S(LG) = \frac{T(p=10,K,1)}{T(p=10,K,LG)}$ for varying lock granularities and different values of K is shown in Figure 4.3.

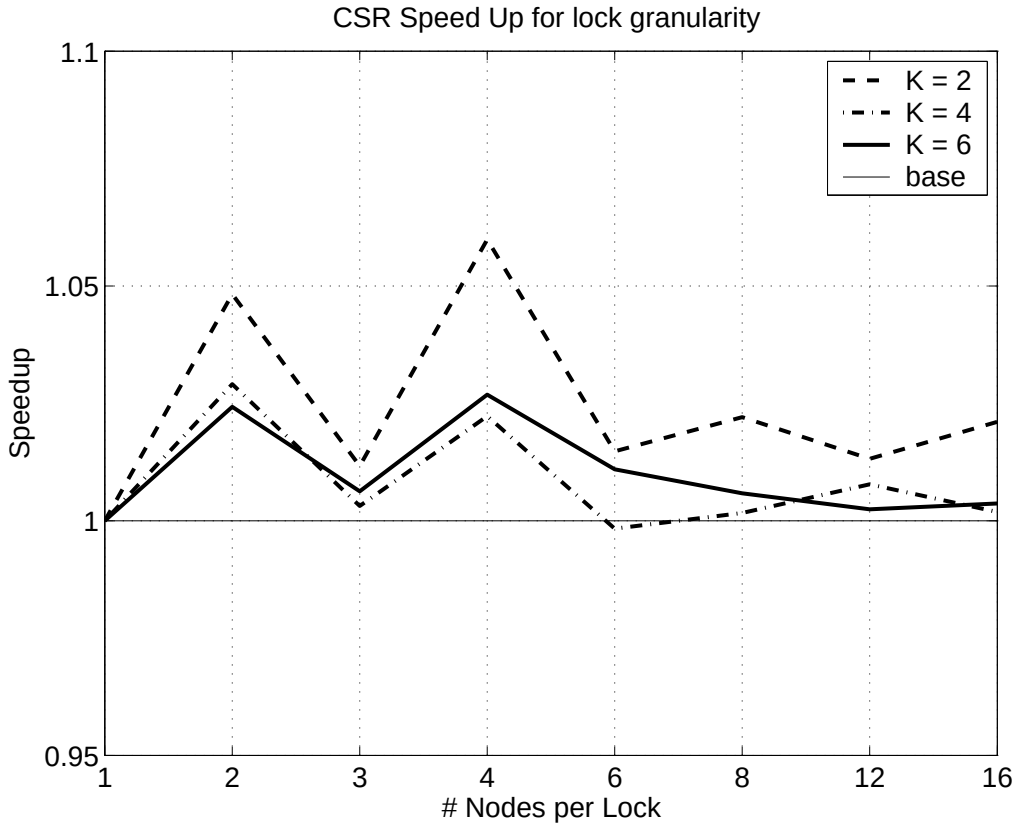

Figure 4.3. Speed up over granularity of 1 for varying lock granularities on the CSR algorithm. 10 processors, 1 million nodes, Ninepoint laplace operator

From Figure 4.3 it can be seen that varying lock granularities have distinct effects on the execution time of the algorithm. The effects however, appear to be random for all lock granularities, except 2 and 4. A first glance it is expected that a power of 2 is the cause of this increased performance, however, the pattern is not repeated for granularities of 8 or 16. At this time, the effect of lock granularity on the CSR algorithm is unknown, and left to future work.

4.1.4 Node removal

Despite the overall efficiency of node removal (see Section 3.4.5), the presence of leftover nodes in the graph results in a minimal amount of extra work, therefore finding a suitable

node removal method was difficult. This section analyzes the best method of node removal, the processor local active list.

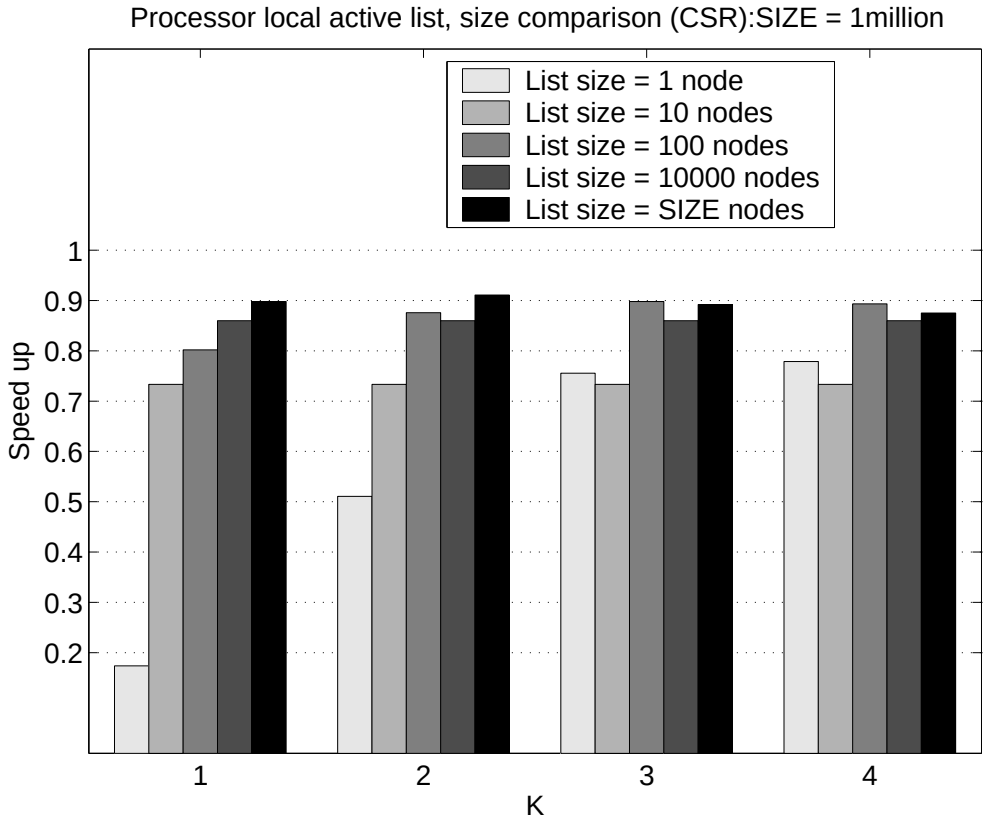

Figure 4.4. Speed up for processor local active list, for various list sizes, vs the base version without node removal.

Processor local active list By using a vector of size N on each processor, a list of active nodes is maintained without the need for explicit synchronization. The initial version of this implementation allowed for a smaller vector per processor. When the vector filled up, a lock is performed and the vector is emptied. However, as seen in Figure 4.4 and discussed below, a list of size N, where locks are not required, is the best choice for this implementation.

Processor local active list experiment This experiment uses a standard graph $D = 1000$ constructed using the standard nine point stencil. The experiment is run with $K = 1 : 4$,

and on 10 processors. $T(K, orig)$ is defined as the basic parallel implementation with no edge or node removal, lock granularity of 1, and no delayed weight updates. The version being examined, the processor local active list, is defined above. The execution time of this version is represented by $T(K, S, local)$. Where S is the size of the active list (values 1 to $SIZE = 1million$ in factors of 10). The speed up $S(K, S) = \frac{T(p=10,K,orig)}{T(p=10,K,S,local)}$ is presented in the bar graph in Figure 4.4

Figure 4.4 shows that there is little difference in performance once the list reaches a considerable size. Therefore, the processor local active list that does not require lock-based synchronization is used. This decision is motivated by the desire to limit synchronization, while keeping the algorithm both simple and efficient.

4.1.5 CSR Edge Removal

In edge removal the overall effectiveness of preventing edge visits is low. Therefore, in order for edge removal to be beneficial, the cost of removing an edge must be sufficiently small. This section discusses the choices made during the implementation of edge removal.

Edge removal In this version, when it is detected that an edge can be removed from the graph, it is set to the proper type, as described in Section 3.4. The edges can be removed in parallel during the second pass when the local maxima are determined. If there is no ordering requirement on the edge list, this removal can be performed in $O(1)$ time. The chosen method for finding intersections, however, requires a sorted edge list; therefore, when removing one or more edges from a list, sorted order must be maintained. This is achieved by using the process described in Section 3.4.4.

Node removal with edge removal A combination of the described version of node removal with that of edge removal yields the complete removal implementation of the CSR algorithm. The only synchronization required by this version, other than synchronization of weight updates, is the use of an implicit barrier between pass 2 and the subsequent pass 1.

Complete removal experiment This experiment is run on on the standard graph $D = 1000$ constructed using the standard nine point stencil. The experiment is run

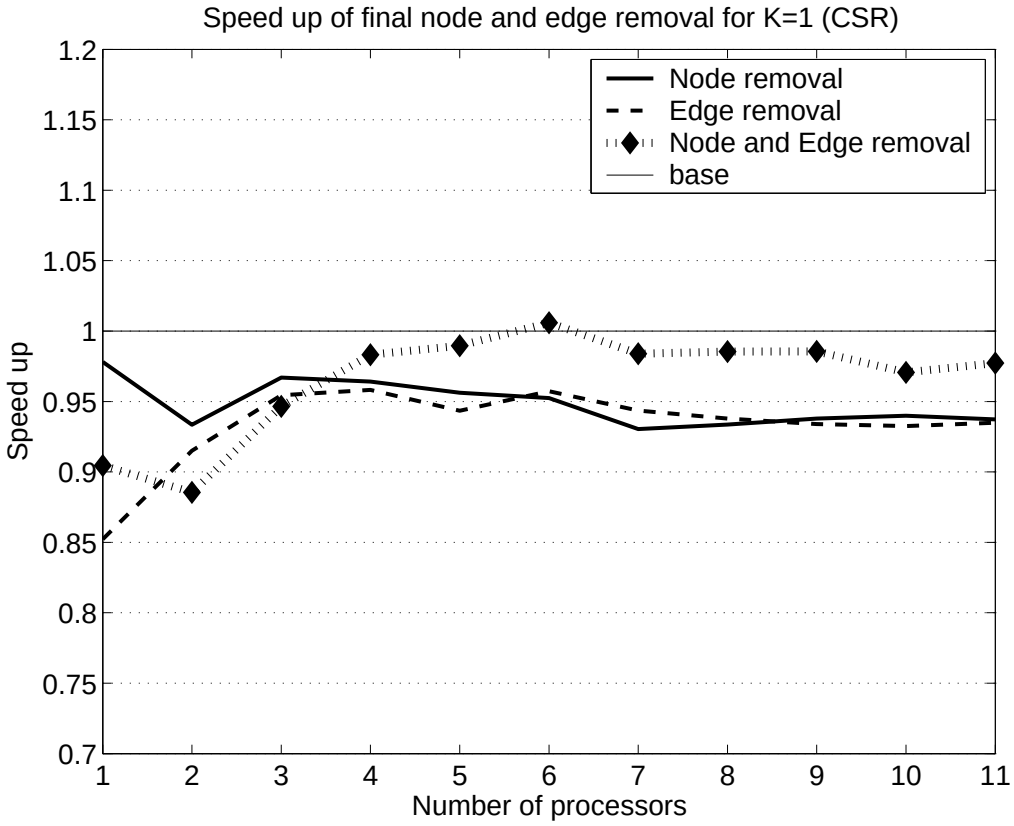

Figure 4.5. Speed up for node and edge removal vs base implementation. Density: K=1

with $K = 1 : 3$, and on 1 to 11 processors. $T(p, K, base)$ is defined as the basic parallel implementation using edge pruning, no node removal, outer loop parallelism, a lock granularity of 1, and no delayed weight updates. The version being examined, the processor local active list with actual edge removal, is defined above. From this version, there are three possible configurations: (1) node removal only, which is defined by $T(p, K, node)$, (2) edge removal only, which is defined by $T(p, K, edge)$, and (3) node and edge removal, which is defined by $T(p, K, node + ege$. The speed ups for these three configurations can be found by using the following equations:

$$S(p, K, node) = \frac{T(p, K, base)}{T(p, K, node)}$$

$$S(p, K, edge) = \frac{T(p, K, base)}{T(p, K, edge)}$$

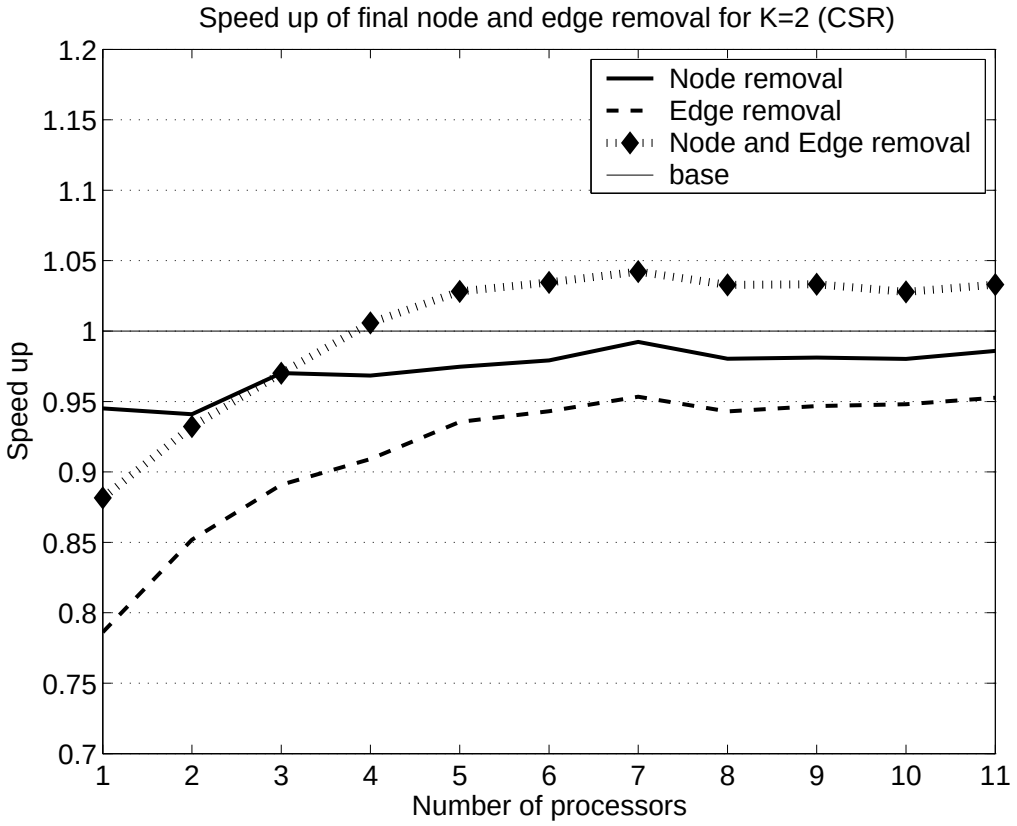

Figure 4.6. Speed up for node and edge removal vs base implementation. Density: K=2

$$S(p, K, node + edge) = \frac{T(p, K, base)}{T(p, K, node + edge)}$$

The results are then presented in Figures 4.5-4.7

From Figures 4.5-4.7 it can be seen that on their own neither node nor edge removal provide any performance improvement at all. In fact, they cause a decrease in performance. However, when combined, node and edge removal show a small increase in performance. Unfortunately this small increase is not a significant one. Regardless, considerable effort was put into determining the source of this small improvement. At the time of writing this thesis, no solution has been found, and thus further investigation must be left to future work.

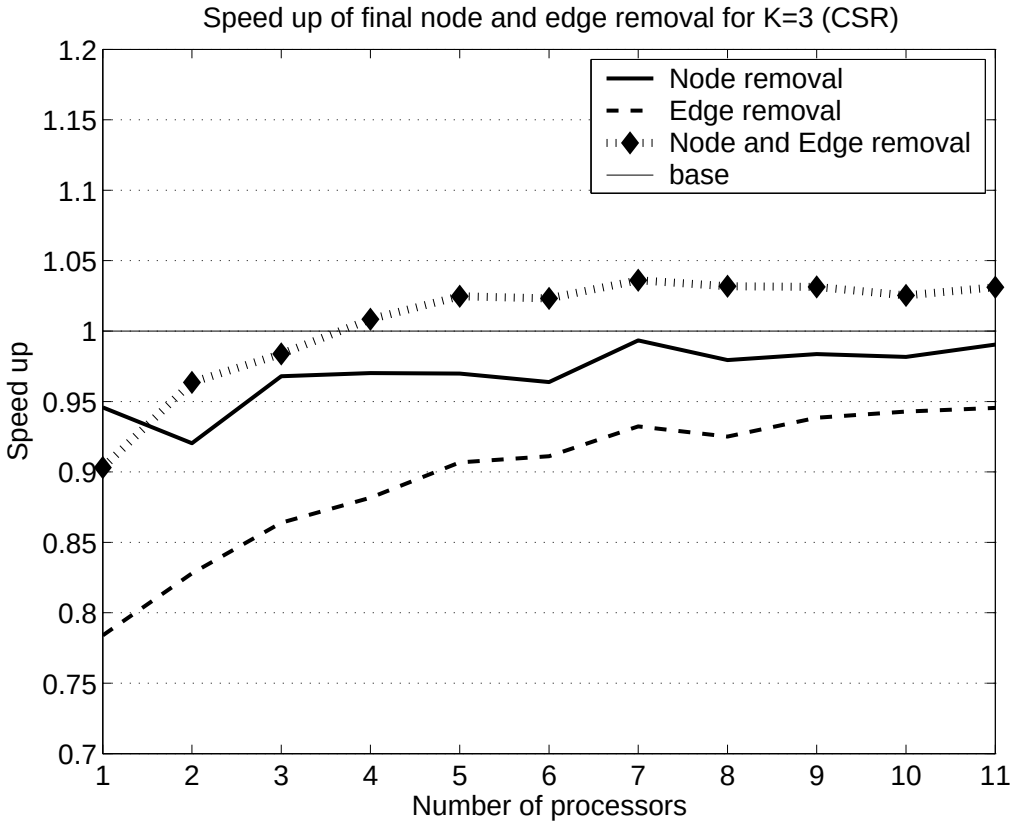

Figure 4.7. Speed up for node and edge removal vs base implementation. Density: K=3

4.1.6 Weight updates

In the row-oriented algorithm weight updates require synchronization. It has been shown, in Section 3.4.8, that the Sun Enterprise E4500 architecture has a significant cost for lock synchronization, even if there is no collision; while barrier synchronization, can provide synchronization for much less. This tradeoff motivated the following modifications to the CSR algorithm, where the weight update locks are either minimized, or completely eliminated

Scatter vector for weight updates By creating a unique identifier for each edge leaving a given node. It is possible to create a temporary $E \times N$ matrix, where E is the number of edges per node, and perform weight updates in parallel. The temporary matrix is indexed by the node id and unique edge id to provide a unique storage location for every possible weight

update. Therefore, weight decrements can be applied to this matrix, in parallel, without the need for locks. At the end of the pass, the weight decrements of each node, stored in the matrix, are coalesced into a single update of the shared weight vector. Since each processor only reads and updates the weights of the node it was assigned, this can also be done in parallel. However, the experiments show that this is a poor choice, as the pass over the vector of size $E \times N$ is too great a cost to be absorbed by the performance increase resulting from the removal of locks.

Hash table for weight updates The poor performance of the scattered weight updates motivated an investigation into a way of reducing the size of the temporary storage. By using a hash table, and exploiting the locality of edge updates, several weight decrements can be collected in a small amount of space. The hash table is implemented with a fixed block size of 2, 8 bit decrement counters. The first 17 low order bits are used for the index (16 bits) and offset (1 bit). The 15 remaining high order bits are used for the tag.

The process of performing a weight update proceeds as follows. When a processor finds that it needs to perform a weight update, it checks its hash table. If the block, to which the weight update maps, is empty then a weight update block is created and one of the decrement counters is increased. If the block contains an entry already, the tag is compared, if it matches then the local decrement counter is updated. Otherwise the processor is forced to perform a lock and update the weight in the shared weight vector.

Experiments This experiment is run on on the standard graph $D = 1000$, constructed using a nine point stencil. The number on processors vary from 1 to 10. Three separate graphs show the results for the three different implementations. The base used for comparison is a basic parallel implementation with no other enhancements, $T(p, K, base)$. The weight vector implementation, $T(p, K, wv)$ is implemented as described above, as well as the inverse of the weight vector, $T(p, K, wv2)$. Finally, the hash table, $T(p, K, wh)$ is also implemented as described. The speedup $S(p, ver)$ is achieved by the following equations:

$$S(p, wh) = \frac{T(p, K, base)}{T(p, K, wh)}$$

$$S(p, wv) = \frac{T(p, K, base)}{T(p, K, wv)}$$

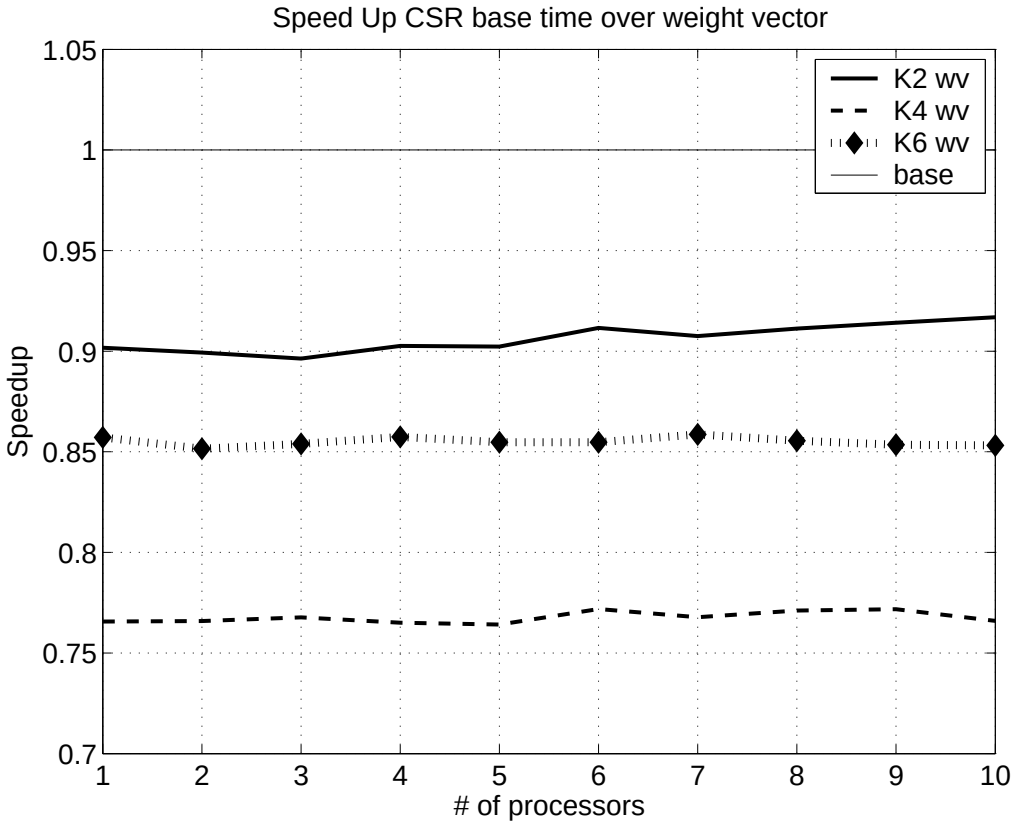

Figure 4.8. Performance of the Weight Vector modification. Speed up for Weight Vector vs. the base implementation.

$$S(p, wv2) = \frac{T(p, K, base)}{T(p, K, wv2)}$$

The resulting $S(p, wh)$, $S(p, wv)$, and $S(p, wv2)$ can be seen in Figures 4.8-4.10. From these figures, it can be shown that that these methods do not provide any improvement in the execution time of the code.

4.1.7 Conclusion

From the analysis of the modifications made to the CSR algorithm, it is apparent that the performance of the algorithm on this architecture, is dominated by the synchronization costs and excessive number of unnecessary edge visits. The experiments in this section show that as long as removed nodes and edges are marked as such, they do not have to be removed

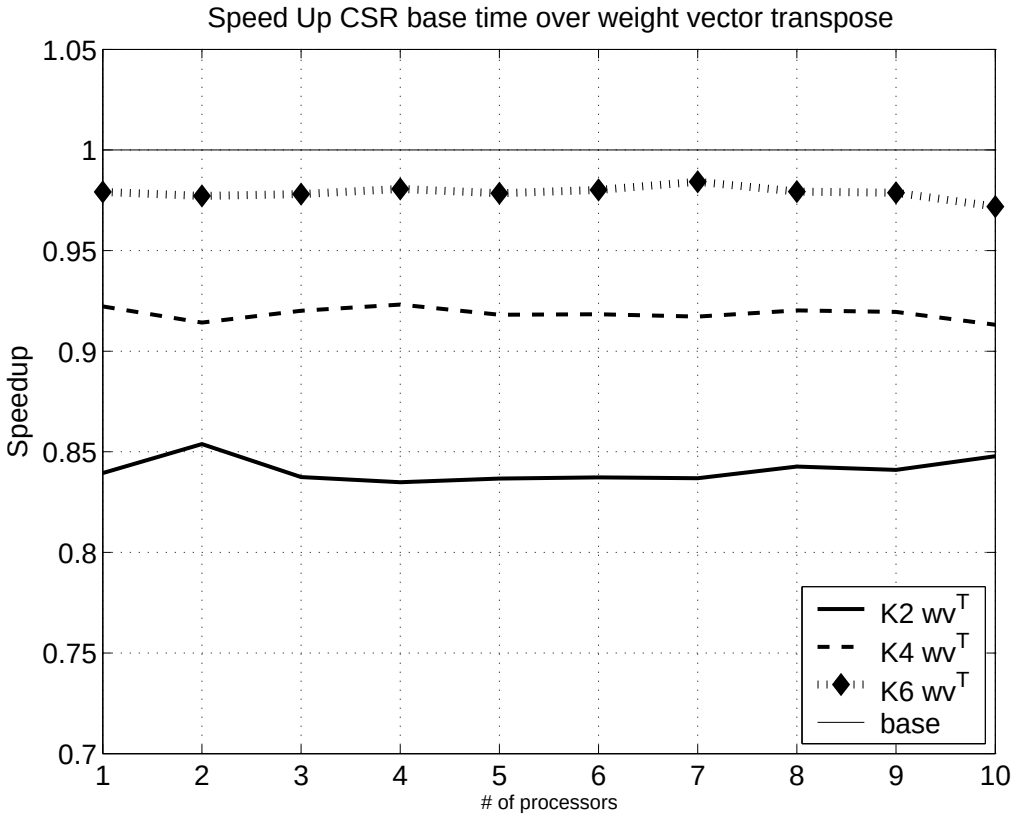

Figure 4.9. Performance of the Weight Vector^T modification. Speed up for Weight Vector^T vs. the base implementation.

from the data structure. As the presence of these marked nodes and edges, does not affect the execution time of the algorithm enough to warrant the extra work and synchronization required to remove them from the graph. Additionally, both the weight vector and hash table did not produce any benefits. It is suspected that the benefit from reduction in number of locks is not great enough to offset the work that is required after the pass. Both of these methods add the equivalent of a third pass over all the nodes in the graph.

Given the experiments in this section the final version of the CSR algorithm is defined as follows. The parallelism is over the outer loop of the algorithm. Node removal is not used, all nodes remain in the graph but are marked as coarse, fine, or undetermined (only the current set of coarse nodes and undetermined nodes are processed). Edge pruning is used to reduce wasteful edge visits. The simplest, lock per weight update locking scheme is used

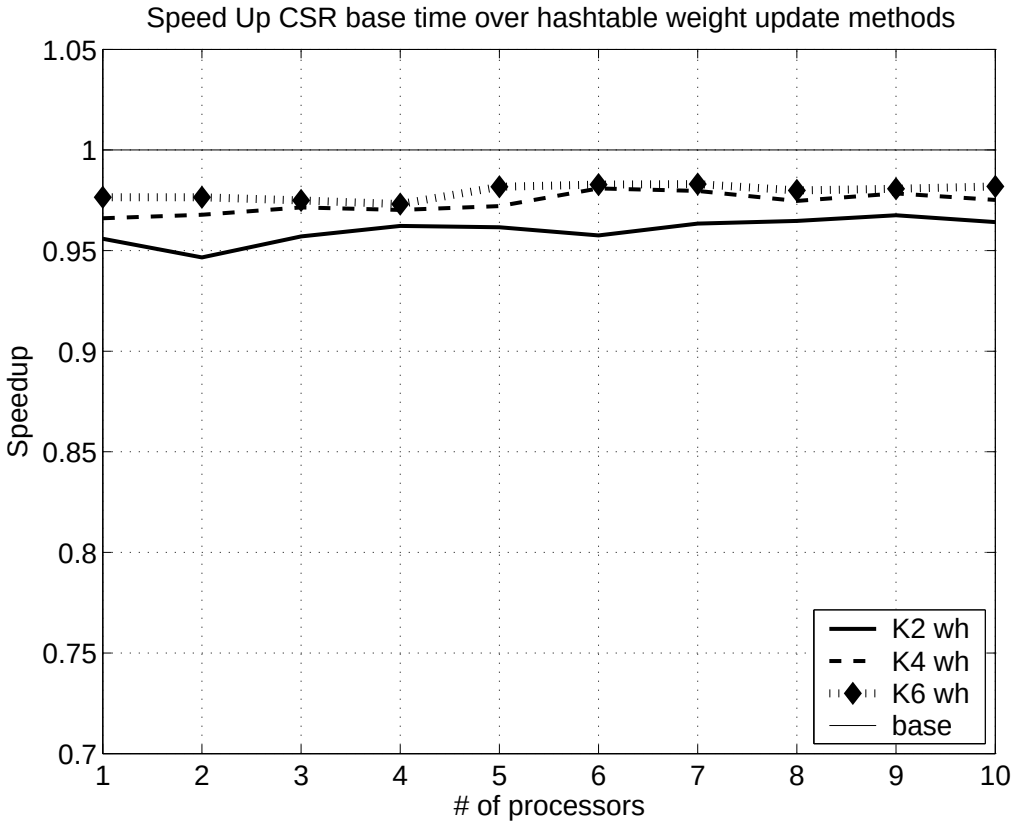

Figure 4.10. Performance of Hash table modification. Speed up for the Hash table vs. the base implementation.

with a lock granularity of 1. This final CSR algorithm is later compared to the other two algorithms defined in Sections 4.2 and 4.4.

4.2 Compressed Sparse Column

This section explores the modifications that can be made to the basic column-oriented version of the CLJP algorithm. This section discusses if and how each of these techniques applied to the CSR-based code can be used to develop a high-performance column-oriented version.

4.2.1 Data structure definition

The CSR data structure had several problems that led to an overall inefficient parallelization. In hopes of avoiding these these problems, a different data structure organization is used. This format is in many ways similar to the previous one (CSR), but it is also has key differences, which allow for a more efficient implementation of the CLJP algorithm. This new format is called *compressed sparse column*, or CSC[8]. The standard CSC format is defined much in the same way as the CSR format is in the last section, except now the non-zero elements of the matrix are stored in column order. In this format AA would be the list of all non-zero elements in the array stored in column order, and once again, just like in CSR, this array filled with 1's and unneeded for the algorithm. JA then would be the row-indices, i in $a_{(i,j)}$, of each element in AA. Finally, the IA array contains the pointers to the start of columns in AA or JA.

For the implementation, the same adaptations that are made in the CSR discussion, are made to this CSC format. The JA array is referred to as S^T, and as before the edge lists of S^T are initially sorted, this time by the value of j in $a_{(i,j)}$. The IA array, now called S^T_ps is augmented with an additional array S^T_pe which points to the end of edge lists. This plays significant role in the column-oriented version of the algorithm, as all Type 2 edges for a given coarse node c can be removed from the edge list by performing the following operation: $S^T\_pe(c) = S^T\_ps(c)$ (i.e. the end of the list is set to the beginning, so this edge list is now empty, just as if all edges are removed by conventional edge removal).

4.2.2 Similarities between the CSR and CSC algorithms

Much of what was learned from the modification of the CSR algorithm can be applied to the CSC algorithm. The following discussions from the CSR algorithm apply directly to this CSC algorithm: (1) the discussion of data structure choices, (2) the level of parallelism discussion, (3) the poor performance of edge removal, and (4) the implementation of node removal.

In the CSC algorithm the data structure is used in much the same way as in the CSR algorithm. Therefore, the performance trend for the array-based data, in the CSC algorithm, is similar to what is observed in the analysis of the CSR algorithm. Additionally,

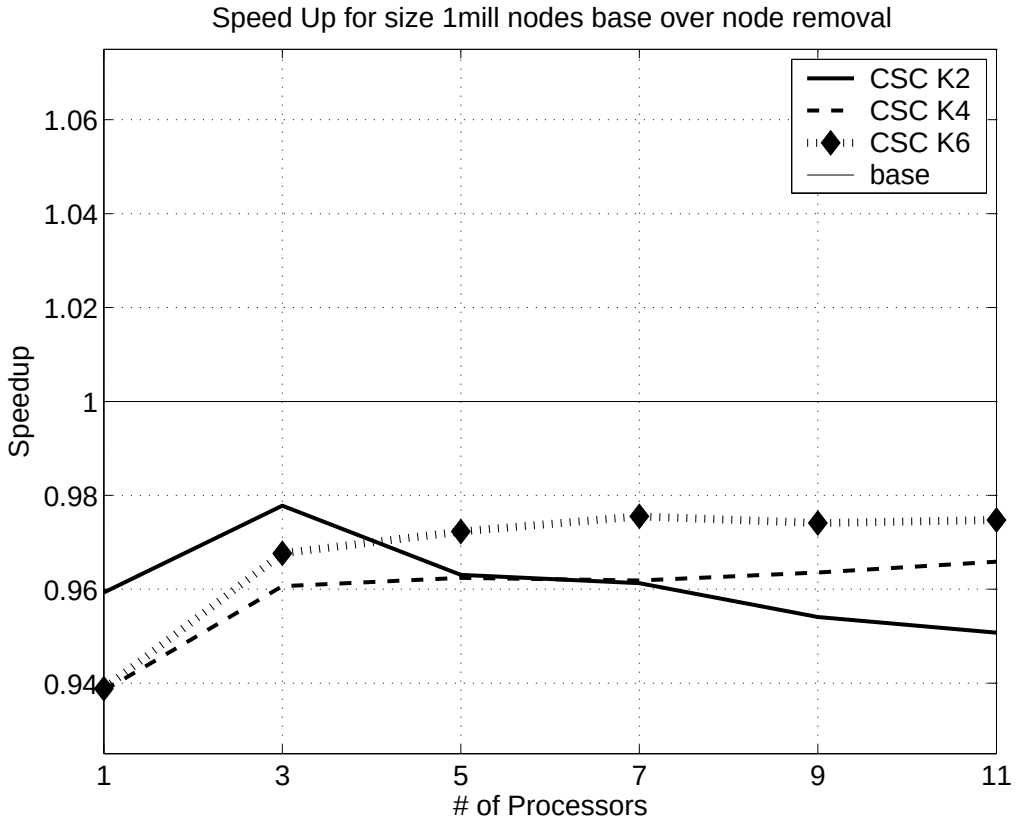

Figure 4.11. Speed up for Node removal (CSC). A simple comparison between the execution time of the CSC algorithm with and without node removal for a number of densities (K). This figure shows that node removal has no significant impact on the execution time of the CSC algorithm.

since the loop structure between the CSR and CSC algorithms is nearly identical, the parallelism discussions map equally well onto the CSC algorithm. Instrumented statistics (Figures 3.9-3.11) show that node and edge removal have the same effectiveness as in the CSR algorithm, making edge removal equally impractical across both versions. Finally, the implementation of node removal is the same as in the CSR algorithm, and as would be expected there are similarities in the performance, see Figure 4.11.

Node removal experiment This experiment uses the standard graph where $D = 1000$ the connectivity is constructed using the standard nine point stencil. The experiment is run with K =2,4, and 6. The number of processors is varied from 1 to 11 in increments

of 2. $T(p, K, base)$ can be defined as the basic parallel CSC implementation using edge pruning, no node removal, a lock granularity of 1, and no delayed weight updates. The version being examined, the processor local active list, is defined in the previous section. The execution time of the processor local active list is defined as $T(p, K, local)$. The speed up $S(p, K) = \frac{T(p,K,base)}{T(p,K,local)}$ is show in Figure 4.11.

From Figure 4.4 it can be seen that, like the CSR algorithm, the CSC algorithm also does not lend itself to node removal.

4.2.3 Differences between the CSR and CSC algorithms

While many of the observations from the CSR algorithm apply to the CSC algorithm, there are two aspects of the implementation that are notably different. These are, fan-in edge traversal and delayed weight updates. First of all, the CSC algorithm, employs a fan-in, instead of fan-out method for weight updates. The fan-in method provides two distinct performance advantages: (1) less wasted edge visits, and (2) improved synchronization.

A large difference in edge visits performed by the CSC algorithm compared to the CSR algorithm results from the method by which the CSC algorithm detects Type 3 edges. In the CSR algorithm, Type 3 edges are detected from the undetermined nodes of the graph. While in the CSC algorithm, because edges are stored based on thier destination node, Type 3 edges are detected from the point of view of a coarse node. By the definition of coarsening, there will typically be a much larger number of undetermined nodes than coarse nodes, and for high densities, the number of undetermined nodes can reach in excess of 80 times the number of coarse nodes. In both versions, the CSR and CSC algorithms, the detection of Type 3 nodes is a complicated process requiring a the traversal of at least $\#\_edges\_per\_node^2$ and at most $\#\_edges\_per\_node^3$ edges. In the CSC algorithm, however, this deep edge traversal only happens for the coarse nodes in the graph, which are not only much fewer in number than undetermined nodes, but also are more likely to drive a useful removal of Type 3 edges.

In addition to decreased edge visits, the CSC algorithm, using a more natural method for Type 1 edge detection, exhibits improved synchronization. In the CSR algorithm a fan-out method is used for edge removal. When processing a coarse node, the CSR algorithm removes all edges leaving that coarse node. This causes the weights of several different nodes to be updated. For each of these weight updates a lock operation is performed. This leads to a high

contention rate, as well as a large overall number of lock operations. The CSC algorithm, however, employs a fan-in approach for edge removal. In the CSC algorithm, coarse-driven weight updates are performed during the processing of undetermined nodes. This allows, in a single pass over the incoming edges, the detection of all Type 1 edges, those with a coarse node as the source. Each of the Type 1 edges causes a weight update, but, unlike the CSR algorithm, in the CSC algorithm these weight updates are all performed on the node that is being processed. This difference provides a number of performance benefits for the CSC algorithm; such as (1) reduction in the number of locks required, (2) better cache performance, and (3) less lock contention.

The reduction in the number of locks comes from the fact that in one pass over the structure S_n^T several Type 1 edges, that each cause an update of $w(n)$, can be detected. By using a temporary counter the weight updates caused by these edges can be accumulated and applied to $w(n)$ after the pass over S_n^T is complete. This only requires a single lock, instead of several, as in the CSR algorithm. At first it would seem that this update does not need a lock. However, because Type 3 edge updates take place on coarse nodes, and affect the weights of undetermined nodes. It can be the case that two processors, even though they are performing Type 1 and Type 3 edge updates, could update the same weight. Therefore a lock is required on Type 1 edge updates as well as Type 3 edge updates.

The fan-in approach also improves cache performance. By only updating a single element of the global data structure less shared cache lines are changed. This means that less cache lines are flushed from the caches of other processors, leading to a greater probability of cache hits.

Lastly, in addition to the reduction in the number of lock operations required, which reduces contention on its own, the fan-in method reduces lock contention directly. Consider the fact that all weight updates driven by Type 1 edges are performed on the node that is currently being processed. Because the sets of nodes given to each processor are disjoint, no two processors performing weight updates based on Type 1 edges can update the same weight. Therefore, there can be no lock contention between two Type 1 driven weight updates. Since Type 2 edges do not cause weight updates, the only possible contention comes from the weight updates of Type 3 edges. This leads to a reduction in the chance of lock contention in the CSC algorithm by $\frac{\#Type1}{\#Type1+\#Type3}$

Unlike the CSR algorithm, delayed weight updates cannot be performed in the CSC algorithm. In the CSC algorithm, while detecting Type 3 edges, two different triangle conditions that would drive the removal of the same edge can be processed at the same time, by different processors. Therefore, a final test of the edge status before the edge is updated is required to prevent the edge from erroneously being marked as Type 3 twice, thus decrementing the weight of its destination node twice. However, even with this final test, it is still possible for one processor to reach the test, when another processor has passed the test, but not updated the status of the edge. In this case, the weight of the destination node is still decremented twice. To solve this problem a lock, indexed by the node-id of the node whose weight is updated, must be acquired before the test and update of the edge status is performed. Thus increasing, relative to the CSR algorithm, the size of the critical section associated with the weight update. While this is not a significant problem for the CSC algorithm, it does prevent any form of delayed weight update from being useful.

4.3 Summary of the CSR and CSC algorithms

Figure 4.12 shows the results of a simple experiment, which is performed to compare the execution time of the CSR and CSC algorithms. As shown in previous discussions, node removal and delayed weight updates have insignificant impact on the execution time of these versions, therefore they are not included in this comparison. The experiment is performed on a graph with $D = 1000$ the connectivity is constructed using a nine-point stencil, with no boundary conditions. The number of processors, as well as the density of the graph is varied to show the performance of the CSR and CSC algorithms under a wide range of conditions. The execution times of the two versions are defined as $T(p, K, CSR)$ and $T(p, K, CSC)$ for CSR, and CSC respectively. The figure shows the speed up $S(p, K) = \frac{T(p,K,CSR)}{T(p,K,CSC)}$, or the number of times faster the CSC algorithm is than the CSR algorithm.

As shown in the previous section, the CSR algorithm has a number of problems. Without deviating too far from the original design of the algorithm a column-oriented (CSC) data structure can be used to greatly simplify the coarsening process. This has been shown in the case of the CSC algorithm. Figure 4.12 shows that the CSC algorithm is up to 15 times faster than the CSR algorithm on one processor, and for $K = 6$ up to 4 times faster on

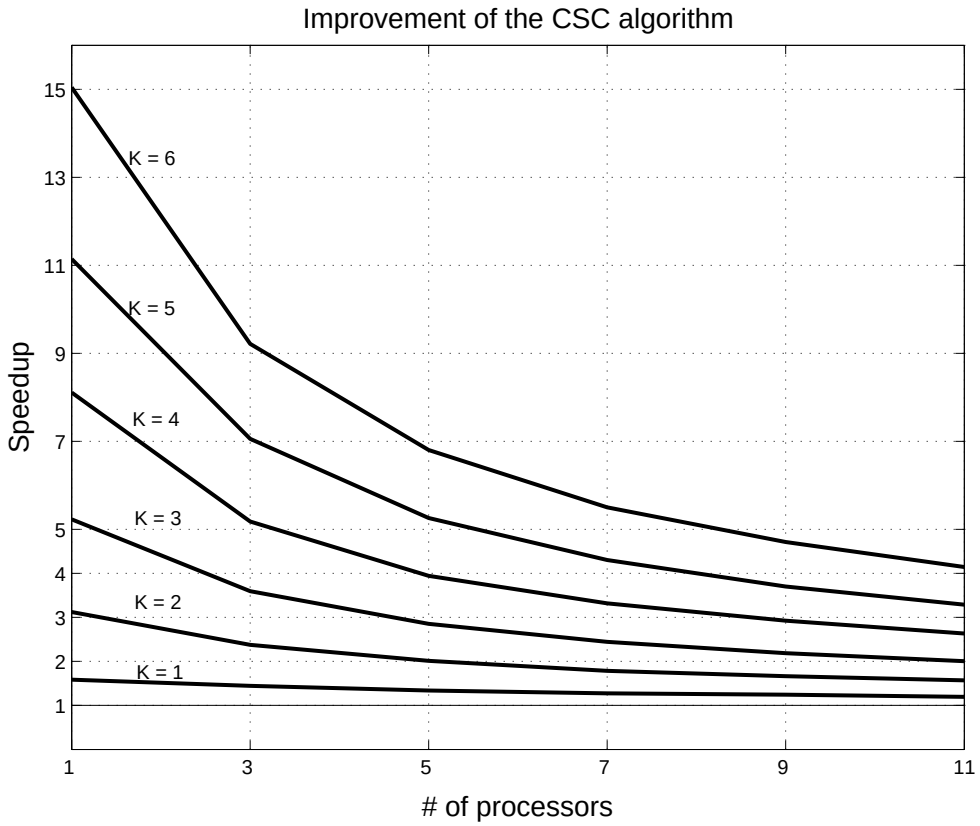

Figure 4.12. Improvement of the CSC algorithm over the CSR algorithm.

11 processors. The downward trend of the CSC algorithm's speed up, as the number of processors increases, is attributed to the parallel efficiency of the CSC algorithm, which is discussed later in Section 5.1

4.4 Modified Version

This section explores the modifications than can be made to the hybrid version of the algorithm, as discussed in Section 3.3 this version uses a combined data structure, called MOD. This data structure has 2 copies of the required edge information. One copy is made up of the row information from the CSR data structure, while the other contains the columns in the CSC data structure. This section discusses how this extra edge information effects the modifications made to the algorithm.

4.4.1 Similarities between CSR, CSC, and MOD

As in the CSC algorithm, much of what was learned in the discussion of the CSR algorithm applies directly to the MOD algorithm. This includes data structure choices, level of parallelism, and poor performance of edge removal. The reasons for these applying across versions is given in the previous section, and is repeated here.

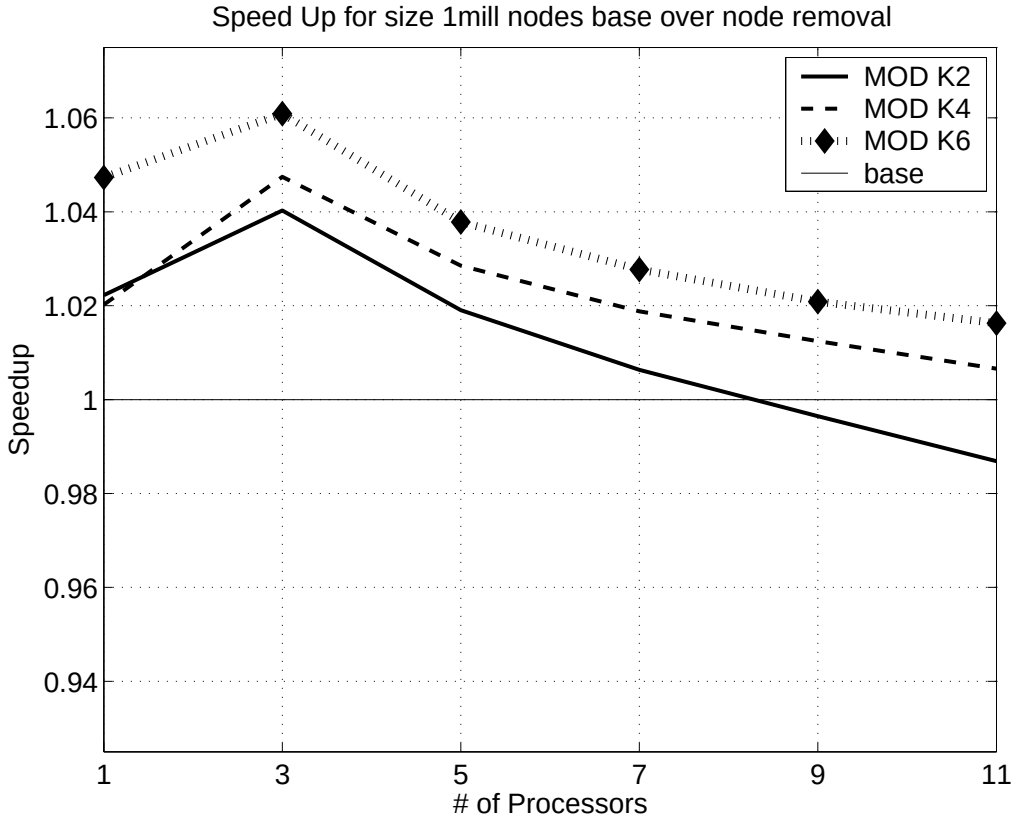

Figure 4.13. Speed up for Node removal (MOD). A simple comparison between the execution time of the MOD algorithm with and without node removal for a number of densities. This figure shows that node removal has no significant impact on the execution time of the MOD algorithm.

In the CSC and MOD algorithms, the data structure, despite storing different data, is used the same way as in the CSR algorithm. Therefore, it is safe to reason that the array-based data structure yields performance trends similar to what is observed in the analysis of the CSR algorithm. Additionally, since the loop structure between the CSR,

CSC, and MOD algorithms is nearly identical, the parallelism discussions map equally well onto the CSC and MOD algorithms. Instrumented statistics in Figures 3.9-3.11 show that, in the MOD algorithm, removal has the same overall effectiveness. Making removal equally impractical across all versions. Finally, in addition to the effectiveness, the implementation and performance of node removal in the MOD algorithm is observed to be consistent with both the CSC and CSR algorithms, see Figure 4.13.

Node removal experiment This experiment uses the standard graph where $D = 1000$ the connectivity is constructed using the standard nine point stencil. The experiment is run with $K = 2$, 4, and 6. The number of processors is varied from 1 to 11 in increments of 2. $T(p, K, base)$ can be defined as the basic parallel MOD implementation with, no node removal, and a lock granularity of 1. The version being examined, the processor local active list, is defined in Section 4.1. The execution time of the processor local active list is defined as $T(p, K, local)$. The speed up $S(p, K) = \frac{T(p,K,base)}{T(p,K,local)}$ is show in Figure 4.13. From this figure it can be seen that, like the CSR and CSC algorithms, the MOD algorithm also does not lend itself to node removal.

4.4.2 Differences between all three algorithms

Many observations apply across all versions, however, there is one important difference between the MOD algorithm and the other two algorithms. In the MOD algorithm, there are no locks required to protect weight updates. This stems from the fact that all information is present to determine the weight updates of each node, from the point of view of that node. Therefore as the algorithm proceeds through the graph, weight updates are only applied to the node that is currently being processed by a particular processor. This allows all weight updates to proceed in parallel.

4.4.3 MOD algorithm performance

Figure 4.14 shows the results from the experiment in Figure 3.12 presented in terms of the increase in number of edge visits between the CSC and MOD algorithms. This figure shows that the while the MOD algorithm has removed all lock-based synchronization from

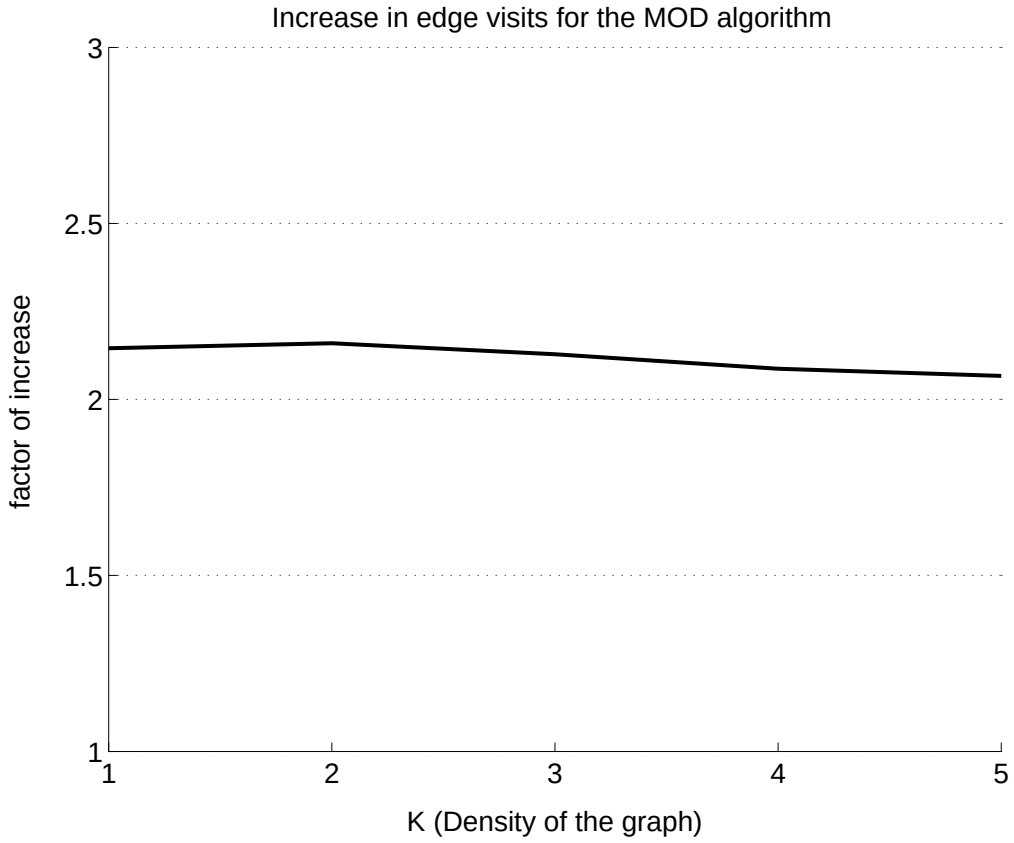

Figure 4.14. Increase in edge visits for the MOD algorithm vs the CSC algorithm. Number shown is how many times more edge visits MOD performs.

the algorithm, it does so at the cost of adding over twice the edge visits. These edge visits, translate to added work, which results in increased execution time.

Figure 4.15 and 4.16 show the results of a simple experiment, which is performed to compare the execution time of all three algorithms, CSR, CSC and MOD. As shown in previous discussions node removal and delayed weight updates have insignificant impact on the execution time of these versions, therefore they are not included in this comparison. The experiment is performed on a graph with $D = 1000$, which is constructed using a nine point stencil. The number of processors, as well as the density of the graph are varied to show the performance of the three versions under a wide range of conditions. The execution of the three versions can be defined as $T(p, K, CSR)$, $T(p, K, CSC)$, and $T(p, K, MOD)$. The speed up, displayed in the figures can be calculated by the following equations:

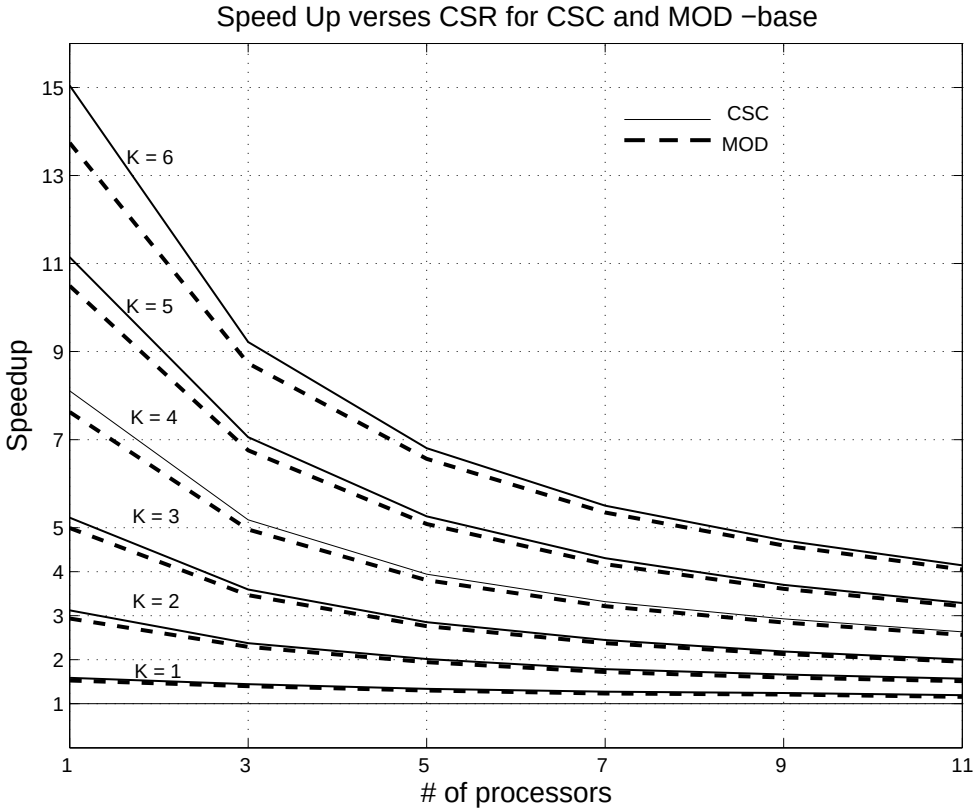

Figure 4.15. Improvement of the MOD and CSC algorithms over the CSR algorithm. Speed ups, $S(p, K, CSC)$ and $S(p, K, MOD)$, show how many times faster MOD and CSC is than CSR.

$$S(p, K, CSC) = \frac{T(p, K, CSR)}{T(p, K, CSC)}$$

$$S(p, K, MOD) = \frac{T(p, K, CSR)}{T(p, K, MOD)}$$

$$S(p, K, CvM) = \frac{T(p, K, MOD)}{T(p, K, CSC)}$$

Note, there is no significant difference in the execution time of the CSC and MOD algorithms. This clearly demonstrates the cost of the increased number of edge visits. The MOD algorithm contains more work than the CSC algorithm, but no synchronization at all. This clear trade-off between the two algorithms is very important. In a system where lock costs are low, the CSC algorithm can be used to reduce total work, at the cost of moderate synchronization. Conversely, in a system where lock costs are very high, the MOD

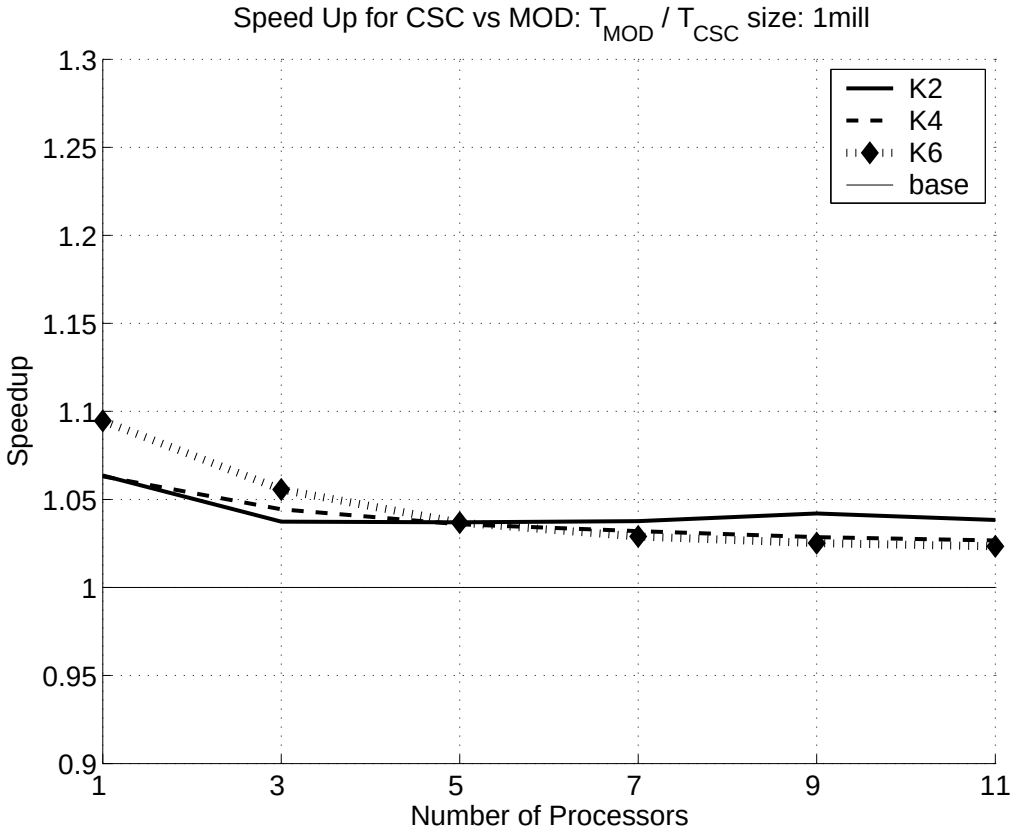

Figure 4.16. Improvement of the CSC algorithm. Speed up, $S(p, K, CvM)$, shows how many times faster CSC is than MOD.

algorithm can be used to eliminate lock-based synchronization altogether, at the cost of over twice the edge visits performed in the CSC algorithm. As the number of processors in a shared memory system grows, the synchronization costs will also increase. For this reason, the MOD algorithm is expected to be the better choice for large shared memory systems.

The MOD algorithm, however, has the opportunity to lead to even better overall execution times. Further optimizations could be explored in the updating of the MOD algorithm's data structure. Because, each copy of the data structure (the CSR and CSC versions) is used to detect specific types of edges, it is possible that these two data structure do not need to be maintained consistently. This could further reduce the number of edge visits performed by the MOD algorithm, leading to better overall execution time, and a significant improvement over the CSC algorithm on the Sun E4500 server.

CHAPTER 5

CONCLUSIONS

This thesis has shown that the choice of data structure used in the implementation of the CLJP algorithm can greatly influence performance. Three data structures were analyzed, this chapter first summarizes the main points of this thesis, explores the parallel efficiency of all three algorithms, presents a number of conclusions, and finally discusses possible topics of future work.

5.1 Summary

The first data structure, chosen due to its widespread use in existing algebraic multigrid solvers, is the *compressed sparse row* or CSR data structure. This data structure stores the information from a sparse matrix in a row-oriented compressed format (storing only non-zero entries). Using the row-oriented CLJP algorithm in [4] as a basis, a shared memory CSR-based algorithm was defined. Because the CSR data structure is the standard for algebraic multigrid solvers, the CSR algorithm was given a extra attention in terms of attempts to improve its performance. The various attempted improvements made to the CSR algorithm include: the removal of unneeded nodes, the removal of unneeded edges, using delayed weight updates to reduce synchronization costs, and various lock granularities. The results of these attempts show that even with a great deal of effort, little actual improvement in performance can be gained over the basic parallel implementation of the CSR algorithm.

The second data structure, chosen as the more natural fit for multigrid coarsening, is the *compressed sparse column* or CSC data structure this data structure stores the matrix in column-oriented compressed format. Using the column-oriented CLJP algorithm in [4] as a basis, a shared memory CSC-based algorithm was defined. It was expected that the fan-in approach of the CSC algorithm would lead to a significant performance increase over the fan-out based CSR algorithm. As Figure 4.15 shows, this was the case, the CSC

algorithm is up to 4 times faster than the CSR algorithm when run on 11 processors. This performance improvement comes from the fan-in nature of the CSC algorithm. Which provides two key improvements, the reduction of edge visits, and a number of improvements in synchronization.

The third data structure, a combination of the first two structures, is the *modified* or MOD data structure. This data structure contains a copy of both of the first two data structures. Using the hybrid CLJP algorithm in [4] as a basis, a shared memory MOD-based algorithm was defined. It was expected that a large increase in edge visits and the elimination of lock-based synchronization would lead to an algorithm with significant trade-offs verses the CSC algorithm. Figures 4.15 and 4.16 show that, this was indeed the case. Despite the large increase in number of edge visits the MOD algorithm has essentially the same performance as the CSC algorithm. This is due to the complete removal of lock-based synchronization from the MOD algorithm. Because the resulting execution time is close to the CSC algorithm, this presents a clear trade off. On systems where the lock cost is relatively high, the MOD algorithm is the best choice, however, on systems where lock cost is low enough the CSC algorithm should be used.

Figures 5.1, 5.2, and 5.3 show the results of a simple experiment, which is performed to determine the parallel efficiency of all three algorithms, CSR, CSC and MOD. Previous discussions have shown that edge removal, node removal and delayed weight updates have an insignificant impact on the execution time of these versions, therefore they are not included in this comparison.

The experiment is performed on a graph with $D = 1000$, which is constructed using a nine point stencil. The number of processors as well as the density of the graph are varied to explore the characteristics of the three versions under a wide range of conditions. In this experiment two different implementations are timed. The first implementation, $T(p, K, x)_p$ is the basic parallel implementation with edge pruning and a lock granularity of 1. The second, denoted by $T(K, x)_{seq}$, is the same implementation but with all synchronization removed. The speed up, $S(K, x) = \frac{T(K,x)_{seq}}{T(p,K,x)_p}$, where x is either CSR, CSC, or MOD, is shown in Figures 5.1, 5.2, and 5.3.

The graphs in Figures 5.1-5.3 show the speed up of the parallel implementation for increasing numbers of processors, and densities. A line with a slope of 1 represents a linearly

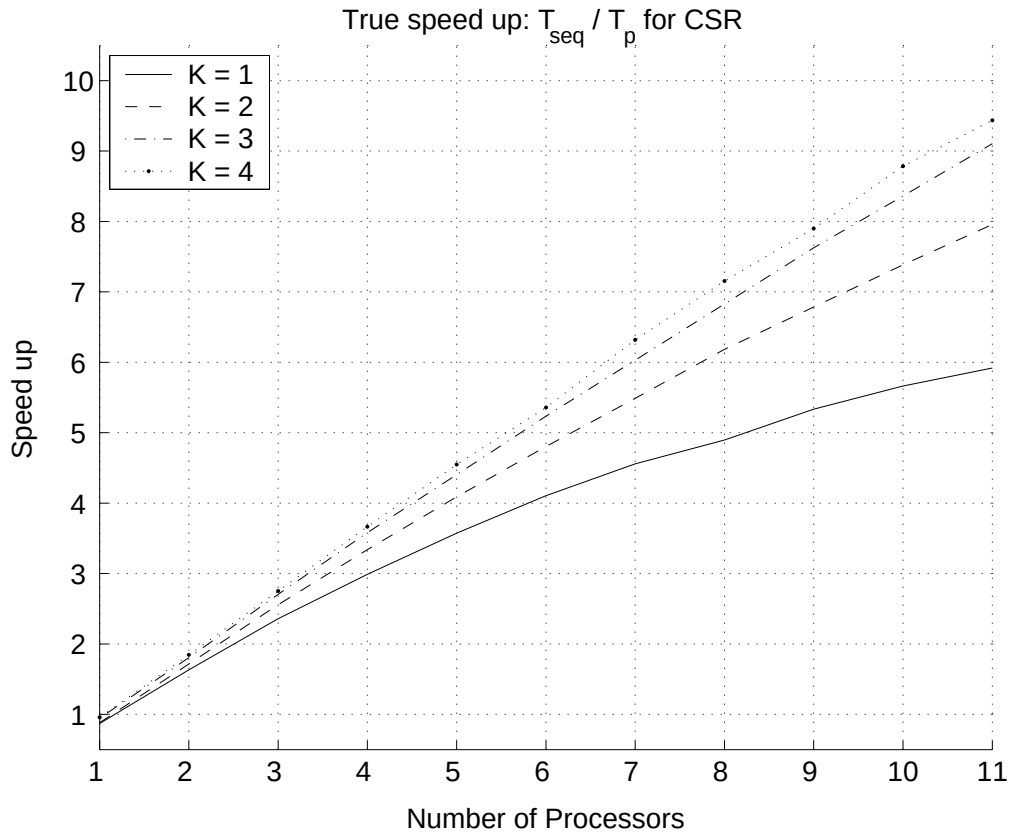

Figure 5.1. True speed up for parallel CSR algorithm, compared to the sequential CSR algorithm (has no synchronization calls).

scalable implementation, which is ideal. In other words, it is desirable for a particular implementation to be 10 times faster on 10 processors than on 1 processor. The closest to this ideal is the CSR algorithm.

In the CSR algorithm the cost of synchronization, which increases with the number of processors, is relatively small compared to the overall work. For this reason the CSR algorithm scales very well with the number of processors.

Conversely, in the CSC algorithm there is much less work, and despite the reduction in lock-based synchronization, the ratio of work to synchronization is much greater than in the CSR algorithm. Therefore the increase in synchronization costs causes a higher percentage increase in total execution time, and thus the CSC algorithm's parallel efficiency is the lowest of all three algorithms.

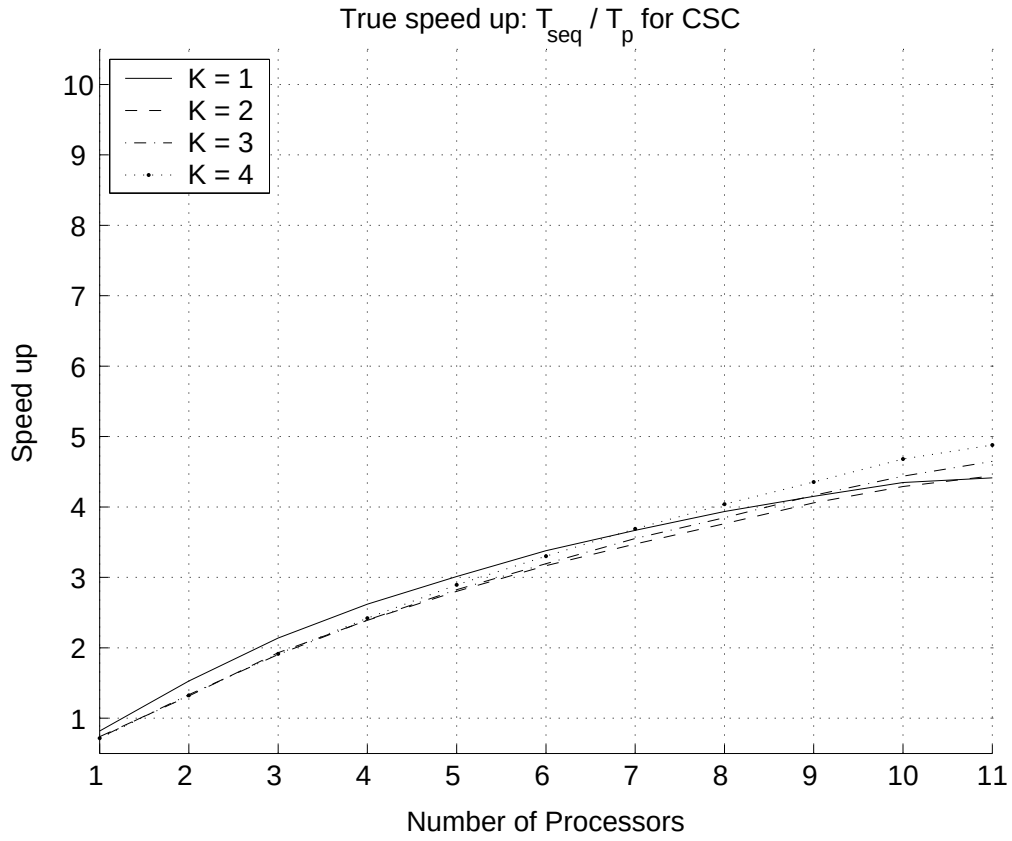

Figure 5.2. True speed up for parallel CSC algorithm, compared to the sequential CSR algorithm (has no synchronization calls).

The MOD algorithm, having the minimum amount of synchronization possible, and twice the work of the CSC algorithm, comes out close to CSR in terms of speedup. The MOD algorithm still lacks the very large amount of parallelizable work that is present in the CSR algorithm, but this is a good thing, since much of that work in CSR is not necessary. This is also why the MOD algorithm does not show as close to linear scalability as the CSR algorithm.

Figure 5.4 shows part of the results from the previous experiment, presented in a different way. The speed up, $S(K, x) = \frac{T(1,K,x)_p}{T(K,x)_{seq})}$, shown in the figure is the parallel overhead of a certain algorithm for varying values of K. As discussed in the previous experiments, the CSR algorithm has a low synchronization to work ratio, and therefore shows a low parallel overhead, especially as K increases. On the other hand, the CSC algorithm, due to its high

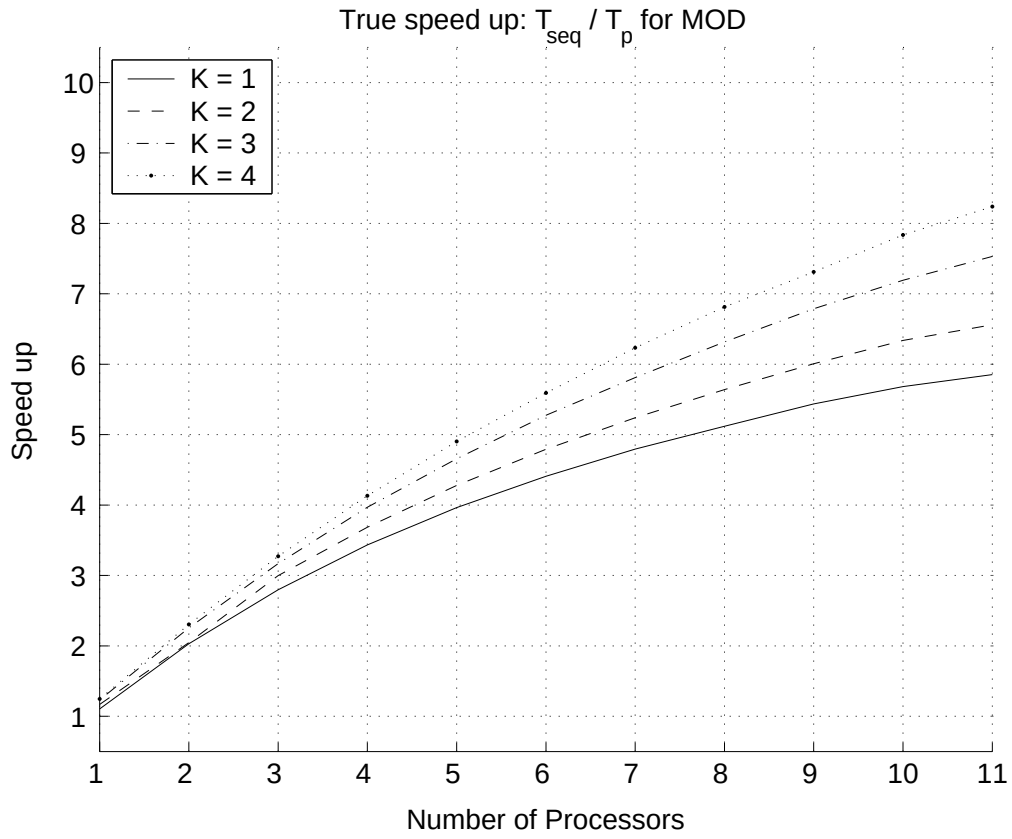

Figure 5.3. True speed up for parallel MOD algorithm, compared to the sequential CSR algorithm (has no synchronization calls).

synchronization to work ratio, shows a high parallel overhead that increases as K increases. Finally, the MOD algorithm shows negative parallel overhead, this is an anomaly. It is suspected that the compiler has performed some optimizations that could not be performed when the code was compiled sequentially. Based on the results in Figure 5.3 and the structure of the algorithm, the MOD algorithm should have low parallel overhead similar to the CSR algorithm.

5.2 Conclusion

In this thesis it has been shown that the influence of the choice of data structure on synchronization is a major factor in the performance of algebraic multigrid coarsening on

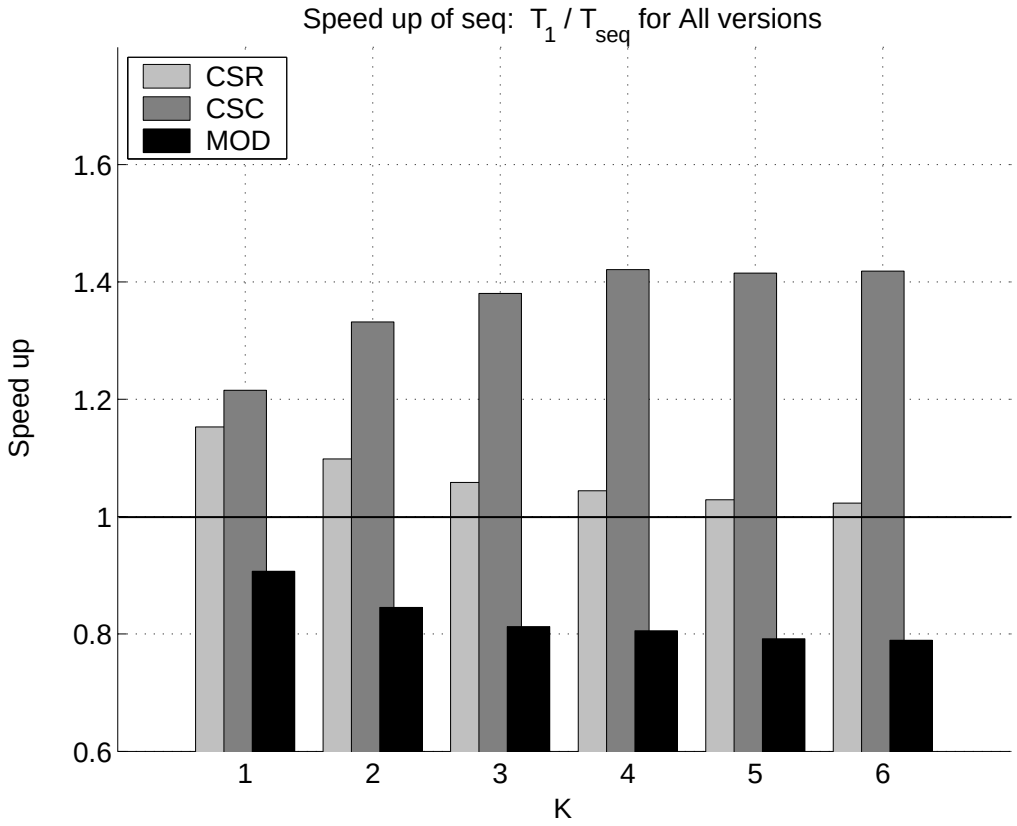

Figure 5.4. Parallel overhead, measured in speed up of the sequential version.

a shared memory architecture. From the execution times observed during this analysis, on large shared memory systems, with high synchronization costs, the MOD algorithm is the best choice. On smaller systems, with lower synchronization costs, the CSC algorithm has a small advantage over the MOD algorithm. The data structure used by the algebraic multigrid solver should also be taken into consideration. If a CSR data structure is used in the AMG solver either a costly conversion process is required, or the solver must be re-written to use a CSC or MOD based data structure. Despite these problems, the improvement in performance gained when switching to a CSC or MOD based algorithm is large enough to warrant exploration of these options.

5.3 Future Work

In the analysis of the three algorithms, CSR, CSC, and MOD, two performance anomalies were encountered. These anomalies are not fully understood, and thus require further investigation. These anomalies are: 1) the interaction of node and edge removal and its affect on performance, and 2) the abnormal performance trends of lock granularity. Finally the possibility of further improving the execution time of the MOD algorithm warrants further investigataion.

The first issue, interaction between node and edge removal, was noticed first while testing the CSR algorithm. Node removal, and edge removal alone lead to a decrease in performance, however, when both are used they cause a small increase in performance. This not only applies to the CSR algorithm, but also the CSC algorithm. This interaction between node and edge removal was investigated when it was first discovered, but the reason behind the synergism could not be found. If this synergism could be understood, it would lead to better understanding of some of the lower level performance effects of these modifications, and the possibility of exploiting them to improve overall performance thus this is a viable are for future work.

The second issue, abnormal performance trends of lock granularity, was also noticed during the testing of the CSR algorithm, but, like the first issue also applies to the other algorithms. In this case, when node removal is not used, lock granularities of 2, 4, and 8 give spikes in performance, but the trend does not continue on to a granularity of 16. Additionally, when node removal is used, the spikes invert, leading to sharp dips in performance at these lock granularities. Like the first issue, better understanding of this performance trend could lead to lock granularity being used to improve overall performance, and is a good source of future work.

The third, and most important area of future work, is the possibility of further improving the execution time of the MOD algorithm. While the MOD algorithm maintains two copies of the required data structures (a CSR and CSC version) it is possible that both copies do not have to be kept consistent. Each copy of the data structure handles the detection of different types of edges. The CSR copy handles Type 1 and Type 3 edges, while the CSC copy handles Type 2 edges. Therefore, which edges actually need to be maintained in which

data structure, should be carefully investigated. If a significant number of edge visits could be prevented in the MOD algorithm, a significant decrease in execution time of the MOD algorithm is expected. This decrease would also show that the MOD algorithm is the best choice for a Sun E4500 server with 11 processors.

REFERENCES

[1] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial* (SIAM, Philadelphia, PA, second ed., 2000) 137-161.

[2] A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, Coarse grid selection for parallel algebraic multigrid, in *Proceedings of the fifth international symposium on solving irregularly structured problems in parallel* (Springer-Verlag, New York, 1998)

[3] K. Gallivan, Private communication, Flordia State University.

[4] K. Gallivan, U. M. Yang, Efficiency Issues in Parallel Coarsening Schemes, Technical Report UCRL-ID-153078, Lawrence Livermore National Laboratory, 2003.

[5] V. E. Henson, and U. M. Yang, BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. Technical Report UCRL-JC-141495, Lawrence Livermore National Laboratory, 2000. to appear in *Applied Numerical Mathematics*.

[6] M. Luby, A simple parallel algorithm for the maximal independent set problem, *SIAM Journal on Computing* 15 (1986) 1036-1053.

[7] J. W. Ruge and K. Stüben, Algebraic multigrid (AMG), in : S. F. McCormick, ed., *Multigrid Methods, vol. 3 of Frontiers in Applied Mathematics* (SIAM, Philadelphia, 1987) 73-130.

[8] Yousef Saad, *Iterative methods for sparse linear systems* (SIAM, Philadelphia, 2003) 89-91.

[9] K. Stüben, An introduction to algebraic multigrid, in: U. Trottenberg, C. Osterlee and A. Schller, eds., *Multigrid* (Academic Press, 2000) 472-478.

[10] K. Stüben, U. Trottenberg, and K. Witsch, Software development based on multigrid techniques, in: B. Enquist and T. Smedsaas, eds., *Proc. IFIP-Conference on PDE Software, Modules, Interfaces and Systems* (Söderköping, 1983).

BIOGRAPHICAL SKETCH

Justin A. Slone

Justin Slone was born on June 1st, 1981, in Palm Beach Gardens Florida. He recently, in Spring 2004, received his bachelors degree, with Honors, from the department of Computer Science at Florida State University. In the Summer of 2004, he began his career in industry at Microsoft.