Parallel Multilevel Graph Partitioning for Shared Memory

Term Project for CSE 392: Parallel Algorithms for Scientific Computing

Makarand Damle Hemanth Kumar Mantri Rahul Thakur Abhishek Tondon

The University of Texas at Austin

{makarand.damle, mantri, rahuljthakur, abhishek.tondon}@.utexas.edu

Abstract

In this term paper, we explore parallel implementations of a multilevel graph partitioner and profile for performance towards the end goal of solving the matrix vector multiplication using Modified Gauss-Seidel solver. We present and compare OpenMP and Pthread implementations for parallelizing each of the phases of multilevel graph partitioning: Maximal Independent Sets (MIS), Graph Coloring, Maximal Matching and Coarsening, followed by Spectral bisection and upward interpolation and finally Modified Gauss-Seidel iterative solver. In this context, we also explore the differences in thread lifetimes and vertex (data) ownership among various threads. We observe that despite very good options for fine-grain synchronization (Pthreads) and task decomposition (OpenMP) offered, the best performance is achieved by preserving data ownership with minimal or no synchronization.

Keywords Graph Partitioning, Graph Coloring, Pthreads, OpenMP, Maximal Matching, Luby's Algorithm, Maximal Independent Sets, Gauss-Seidel solver

1. Introduction

Divide and Conquer strategies are universally accepted as a standard solution to large-sized problems. Graph partitioning is a manifestation of divide and conquer approach applicable on very large graphs. The problem involves dividing the vertices of a graph into groups of roughly equal sizes while taking care to ensure that the number of edges connecting the vertices belonging to one group to those to other groups are minimal. This problem is an NP-complete problem, and hence needs iterative, heuristic methods for solving it. In this paper, we employ a multilevel approach as proposed by LaSalle and Karypis [1]. The steps include:

- finding maximal independent sets of the original graph
- coloring the graph
- perform maximal matching and coarsening
- do spectral bisection of coarsened graph
- partition original graph based on spectral bisection of coarsened graph

The coarsened graph has fewer vertices and is deemed as the next level graph. At each level, same set of steps are repeated until we are left with a graph having a small number of vertices. Thereafter, spectral bisection is employed multiple times to bisect the graph into two clusters of vertices, and upward interpolation is employed to get original graph partitioned into a number of smaller ones. Usually, this entire exercise of graph partitioning is a means towards the end goal of solving a larger problem. In this paper, the problem being solved is : *matrix vector multiplication*. Mat-Vec multiplication, when solved by iterative solvers such as *Jacobi or Gauss Seidel*, requires several iterations for very large graphs and hence is very slow. While it is possible to parallelize this problem in a naïve manner by letting the threads take their share of vertices and work on them in parallel, that results in cross-thread communication and lack of data ownership and locality. By performing the additional exercise of graph partitioning to have the large original graph partitioned into smaller graphs, each of which can be owned by a separate thread. They can all be solved in parallel with minimal cross-thread interaction. This forms the key-idea of our project. We use *OpenMP and Pthreads* to have two competing versions of the implementation and compare them both against the naïve-partitioned modified Gauss-Seidel solver's performance.

In this paper we make following contributions:

1. Implement *OpenMP* based graph partitioner-cum-solver
2. Implement *Pthreads* based graph paritioner-cum-solver
3. Compare both implementations with a naïve *OpenMP* based Modified Gauss-Seidel solver.
4. Compare both implementations with a naïve *Pthreads* based Modified Gauss-Seidel solver.

The rest of this project report is organized as follows. In section 2 we provide a definition of the graph partitioning problem and mention some existing well known graph partitioners. We explain in detail our implementation of a multilevel graph partitioner in section 3. Section 4 shows our experimental evaluation and analysis. The lessons learned and the hurdles faced in course of this project are discussed in section 5 and conclude in section 6.

2. Definition and Related Work

A graph, denoted by *G = (V, E, W)* consists of a set of vertices *V* and a set of edges *E*, where each edge *e* = {*u,v*} contains a pair of vertices (*i.e., v, u in V*). Each edge $e \in E$ has a positive weight $w \in W$, associated with it. Also, each vertex $v \in V$ may have a weight associated with it. However, when no weights are associated with the vertices and/or edges, then their weights can be assumed to be one.

For any vertex $v \in V$, its set of adjacent vertices are denoted by *adj(v)*. A partitioning of *V* into *n* mutually exclusive and exhaustive

subsets $V_1, V_2, V_3 ..... V_n$is called a n-nway partitioning of graph and each set V_i is referred to as a partition.

Edges that connect vertices in different partitions are considered to be cut by the partitioning and the sum of the weight of the cut edges is called the *edgecut* of a n-way partitioning. Most graph partitioning algorithms try to optimize (usually minimize) the *edgecut* However, we partition the given graph not aiming at minimizing the *edgecut* but to achieve near equal number of nodes in each partition which shall be worked on by a separate thread. This ensures that each thread gets nearly same amount of work and the communication overhead is minimized which means no or minimal synchronization.

Typical *Multilevel partitioning methods* consist of three distinct phases:

- Coarsening
- Initial partitioning
- Uncoarsening(Interpolation)

In the *coarsening* phase, the original graph *G0* is used to create a series of increasingly coarser graphs, $G_1, G_2, ...G_m$. In the *initial partitioning* phase, a partitioning P_m of the much smaller graph G_m is generated. Lastly, in the *uncoarsening* phase, the initial partitioning is used to derive partitionings of the successive finer graphs. This is done by first *projecting/interpolating* the partition of $G_i + 1$ to G_i.

Several approaches for graph partitioning have been developed and evaluated [4]. *Coarsening* is usually performed by computing a *matching* [3, 5] or *clustering* [6] through approaches based on weighted aggregation have also been explored in [7, 8].

Our approach is much similar to *KMetis* [12] which stores the graph in a structure similar to that of a Compressed Sparse Row format used for sparse matrices. *ParMetis* [13] is an MPI based parallel formulation of *KMetis* and as a result achieves parallelism through *p* processes.

3. Implementation

We implemented two versions of multilevel parallel graph partitioner-cum-solver: *Pthreads* based and *OpenMP* based. In addition, we also implemented *Modified Gauss-Seidel* iterative solver with naïve work partitioning, also with Pthreads and OpenMP.

Algorithm 1 shows some pseudo code which forms the basis for all versions of our graph partitioner-cum-solver.

```
Input  : A weighted-edge-list E (u, v, w), #vertices,
         #edges, num_threads, coarse_level
Output: Solution of Ax = 0 where A is the adjacency
         matrix from input graph
foreach edge e ∈ E do
  | gr = populate_graph(e);
end
while i++ < num_threads do
  | while num_vertices > coarse_level do
  | |  parallel_mis(gr[i]);
  | |  parallel_color(gr[i]);
  | |  parallel_maximal_match(gr[i]);
  | |  parallel_coarsen_graph(gr[i]);
  | end
  | parallel_spectral_bisect(gr[i]);
  | parallel_interpolate(gr[i]);
end
parallel_ModGS(gr, num_threads);
```

Algorithm 1: Multi-level Graph Partitioning Algorithm

3.1 Pthreads Setup

A pool of threads (number is a command-line option) is created during initialization. Each thread waits for some *work* to be assigned and once available, starts executing it. Every operation that can be run in parallel on each vertex/component, such as certain steps of *MIS, coloring, matching, spectral bisection* is packaged as a *work item* which will be picked by the corresponding thread.

Since most of the *work* being done by a thread is restricted to its own *share* of vertices/components, we minimize inter-thread synchronization and communication. However, we still need very minimal barrier synchronization across iterations, sometimes. Algorithm 2 shows the function executed by every thread.

```
Input: Work Queue (wq)
while (1) do
  | sem_wait(work_item);
  | w = dequeue(wq);
  | w → func();   //execute the function
  | if kill_threads then
  |  | thread_exit();
  | end
end
```

Algorithm 2: Pseudo Code for each thread

3.2 Parallel MIS

The major steps involved in our standard shared memory implementation of parallel MIS algorithm are:

- In each iteration assign a random number to each vertex.
- Insert a vertex to MIS *iff* it has the largest number of all its neighbors.
- Remove all neighbors of the vertex inserted into MIS.

3.3 Parallel Coloring

Our shared memory graph coloring algorithm uses MIS. The major steps in the algorithm are:

- Run parallel MIS
- For each vertex in MIS, assign smallest color not taken by neighbors in parallel
- Remove vertices in MIS from the original vertex list
- Repeat until original vertex list is empty

3.4 Parallel Maximal Matching

Once we have a colored graph, maximal matching is done. All vertices are initially unmatched and we have one queue initialized per color. Algorithm 3 shows how matching is done for a given colored graph.

3.5 Parallel Spectral Bisection

Spectral bisection is used to split the coarsened graph into two sets of positive and negative vertices. The major steps in the algorithm are as follows:

- Use LAPACK [14] to find second *Eigen value* and thus second *Eigen Vector*
- Use second eigen vector to split coarsened graph into positive and negative disjoint sets of vertices.

Input : Colored Graph G(V, E), num_colors
Output: Maximal Matching of G

foreach *vertex v ∈ V* **do**
  match(v) = -1;
end
foreach *color c ∈ num_colors* **do**
  q = queue_init(); **foreach** *vertex u ∈ V such that color(v) = c and match(v) = -1* **do**
    v = vertex in adj(v) with highest weight;
    match(u) = v; match(v) = u; queue_add(u)
  end
  foreach *vertex u ∈ queue* **do**
    if (u ! = match(match(u))) match(u) = -1
  end
end

Algorithm 3: Maximal Matching Algorithm

3.6 Parallel Interpolation/Projection

In this step, we map the positive and negative disjoint sets of vertices obtained from parallel spectral bisection and interpolate them back to the original graph. This is then med to split the original graph into two partitions. Major steps in the algorithm are as follows:

- Map all negative vertices of the coarsened graph to its original uncoarsened starting graph and construct positive graph A.
- Map all positive vertices of the coarsened graph to its original uncoarsened starting graph and construct positive graph B.

3.7 Parallel ModGS

In the main implementation where multilevel graph partitioning results in the original graph being partitioned into as many smaller graphs as the number of threads (again, OpenMP or Pthreads), each thread works on the entire (smaller) graph. Error threshold remains the same for each graph, and so does the limit of maximum iterations. When all threads meet the level of threshold, all threads break out together. As long as even one thread hasn't met the threshold, all threads have to keep running. This is necessary because to maintain correct operation, we need to have a barrier where all threads have to synchronize before a new iteration begins. Since a barrier must be executed by every single thread, before any thread can go past the barrier, that's why all threads have to do same number of iterations, even if some of them converged to threshold before the others.

3.8 Benchmark Solver

Our multilevel-partitioned solver (for AX = 0) will be compared against a basic iterative solver (modified Gauss-Seidel), in which the work partitioning is done without any consideration for locality. Within each iteration of our modified Gauss-Seidel solver, the vertices are equally divided among threads (either *OpenMP* or *Pthreads*), and each thread acts on its own share. Each thread can read vertex data owned by other threads but can write only to its own share. This ensures minimal synchronization. Because only the neighbors are stored for each vertex, while performing modified Gauss-Seidel operation, only their edge weights need to be checked, instead of iterating over entire vertex list. This greatly reduces the number of iterations. The vertex structure keeps an ID which helps getting the position in the actual adjacency matrix, which is required to perform the multiplication of the weight with the right corresponding X[] element. The root mean square error is calculated at each iteration and solver goes on until either it reaches the upper limit of number of iterations or the error goes below the threshold. In our implementation, we set an error threshold of 0.0001 (error norm between iterations) and the maximum iterations to 100000.

4. Evaluation

4.1 Hardware Description

All our experiments and profiling mentioned in this section were conducted on TACC's *Stampede* utilizing a single socket on a node. Each socket is a Intel Xeon E5-2680 2.7 GHz with maximum 16 threads with 20 MB L3 cache with 32 GB distributed memory per node.

4.2 Results and Analysis

Figure 1 shows strong scaling behaviors for naïve-partition and our Multilevel-partitioned implementations. The numbers indicate execution times for solving the entire New York road network graph consisting of 264346 vertices and 366923 edges. The time reported below are averaged over 5 runs. Further, for the multilevel-partitioned solver the execution time includes:

1. Time taken for Multi-level based partitioning
2. Each thread independently solving its assigned partition

It can be seen that the there is about 10-30% reduction in the execution times when multilevel-partitioning is used. We found that across different runs, based on the outcome of spectral bisection and coarsening operations, slightly different times are observed and therefore we present times obtained by averaging over 5 runs in all our results.

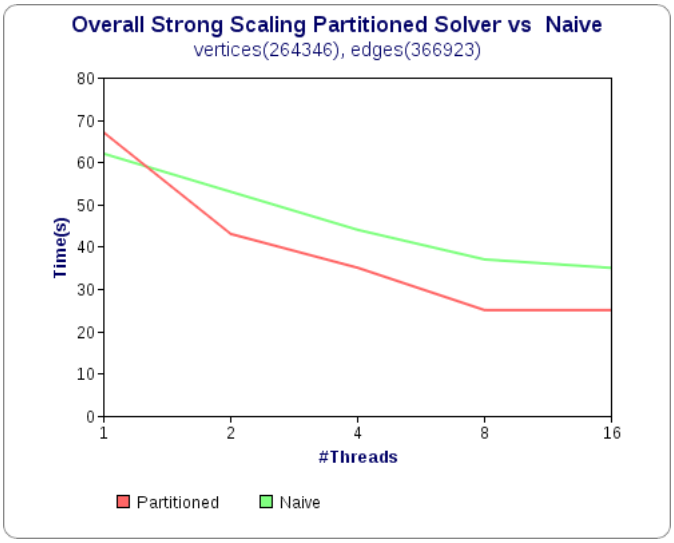

Figure 1. Strong Scaling Multilevel-Partitioned vs. Naïve-Partitioned solver

From our findings the edge cuts observed for a binary tree graph of depth 13 containing 16384 vertices on creating 16 partitions was 80. This proves that doing spectral bisection operation was able to better partition the graph keeping very high topological locality within the partition.

Parallelizing using Pthreads gives around 5-8% improvement in execution times with respect to OpenMP implementations of our multilevel-partitioned solver. This is show in figure 2.

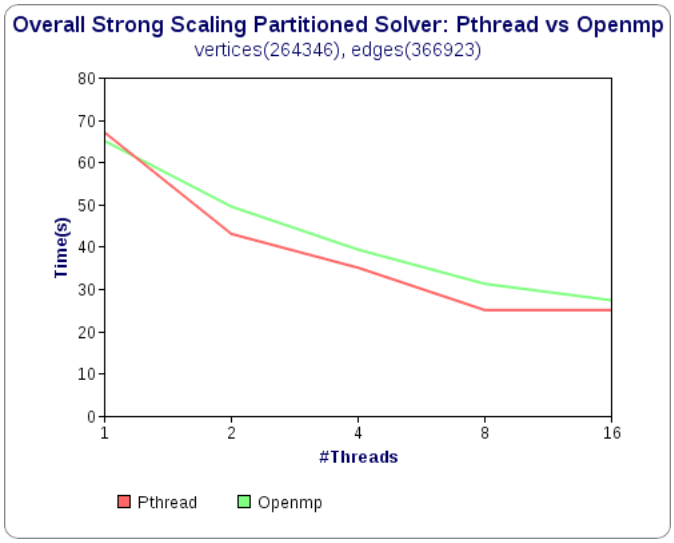

Figure 2. Strong Scaling Partitioned Solver OpenMP vs. Pthread

In order to quantify the cost involved in the memory-bound coarsening operation, we present strong scaling results obtained by a single coarsen operation in figure 3. This coarsening operation was done on the same New York road network graph to reduce it to a coarsened graph having less than 1000 vertices.

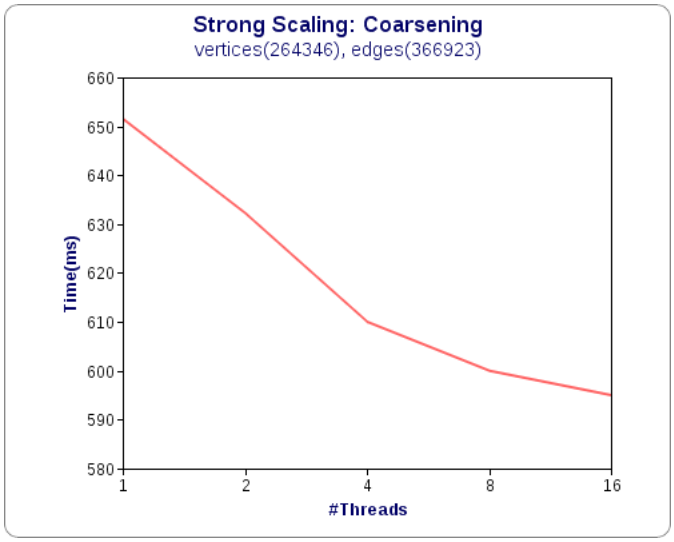

Figure 3. Strong Scaling: A Coarsening operation

Figure 4 shows the execution time spent in MIS operation for first iteration of coarsening of the sparse New York Graph. We observe very close to linear scaling which goes sub-linear for 8 and higher core counts due to the machine used (single socket on a node on Stampede).

Figure 5 shows weak scaling behavior of our multilevel-partitioned solver. The execution time scales nearly giving a flat weak scaling profile.

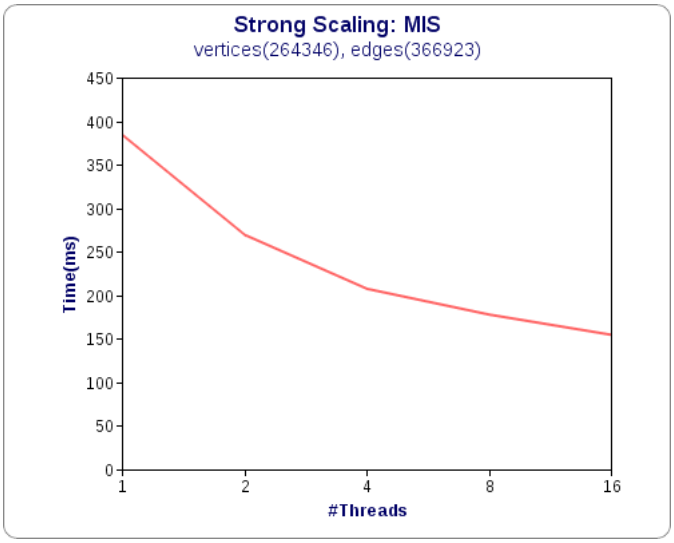

Figure 4. Strong Scaling: A MIS computation operation

Since, the total effort involved in coarsening and spectral bisection phases increases with an increase in the number of threads, the time taken by those phases increases as well. However, if the effort is kept constant (i.e. made invariant wrt. number of threads), then, with an increase in number of threads, the time taken by those phases decreases.

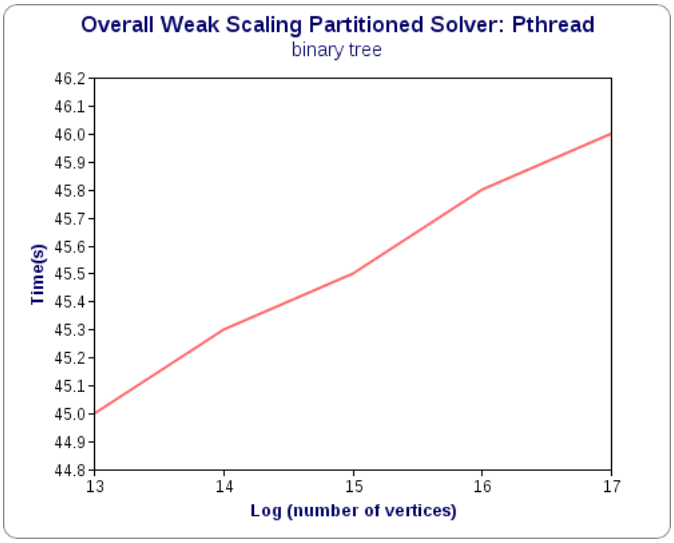

Figure 5. Weak Scaling: Partitioned Solver Pthreads

5. Lessons Learned

In course of this project, we faced a few hurdles and learned some lessons.

- During the course of our implementation, while profiling different code segments, we discovered that the probability based selection approach of Luby's algorithm in the MIS construction step proved to be excruciatingly expensive. It contributes to more than 99% of the total time consumed in MIS construction.

Therefore, we switched to standard shared memory MIS construction which resulted in substantial reduction in MIS construction step's execution time.

- Another lesson in place was with regard to the handling of creation and destruction of Pthreads. Our first version created and destructed threads on need basis. This had resulted in a significant overhead. Therefore, we modified it to a queue based implementation where threads were all created and destroyed only once.
- For the Spectral bisection and Eigen Solver step, we switched from FEAST [15, 16] solver to LAPACK [14] solver. FEAST is based on a numerical method, in which the solution Eigen values are searched within a specified range. Thus, due to the inherent variability of the problem, the results varied for a given problem. This issue was fixed with introduction of LAPACK based SSYGVX solver.

6. Conclusion

One of primary motives of the project was to explore how multilevel partitioning of large graphs facilitates quicker solving of matrix vector multiplication in comparison to native task partitioning among parallel threads.

As the results indicate, for different number of threads and different graph sizes, even with the cost of multilevel partitioning included, the total time taken to solve graphs decreases as compared to naïve iterative solver approach.

1. For multilevel graph partitioning, Pthread implementation works slightly better than OpenMP.
2. Both OpenMP and Pthread implementations shows similar scaling behavior.
3. As the number of threads increases, the time taken by the solve phase decreases.

References

[1] "Multi-Threaded Graph Partitioning", Dominique LaSalle and George Karypis, 27th IEEE International Parallel and Distributed Processing Symposium, 2013

[2] Luby, M.: "A simple parallel algorithm for the maximal independent set problem", SIAM J. Comput. 15(4), 10361055 (1986)

[3] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," Sandia National Labratories, Tech. Rep. SAND93-1301, 1993.

[4] K. Schloegel, G. Karypis, V. Kumar, J. Dongarra, I. Foster, G. Fox, K. Kennedy, A. White, and M. Kaufmann, "Graph partitioning for high performance scientific simulations," 2000.

[5] G. Karypis and V. Kumar, "Multilevel graph partitioning schemes," in ICPP (3), 1995, pp. 113122.

[6] "Multilevel k-way hypergraph partitioning," in Proceedings of the 36th annual ACM/IEEE Design Automation Conference, ser. DAC '99. New York, NY, USA: ACM, 1999, pp. 343348.

[7] I. Safro, D. Ron, and A. Brandt, "Multilevel algorithms for linear ordering problems," J. Exp. Algorithmics, vol. 13, pp. 4:1.44:1.20, Feb. 2009.

[8] C. Chevalier and I. Safro, "Learning and intelligent optimization," T. Stutzle, Ed. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Comparison of Coarsening Schemes for Multilevel Graph Partitioning, pp. 191205.

[9] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," The Bell system technical journal, vol. 49, no. 1, pp. 291307, 1970.

[10] C. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in Design Automation, 1982. 19th Conference on, june 1982, pp. 175 181.

[11] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. F. Werneck, "Graph partitioning with natural cuts," in IPDPS. IEEE, 2011, pp. 11351146.

[12] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," Journal of Parallel and Distributed Computing, vol. 48, pp. 96129, 1998.

[13] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs, in Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM), ser. Supercomputing 96. Washington, DC, USA: IEEE Computer Society, 1996.

[14] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. 1990. LAPACK: a portable linear algebra library for high-performance computers. In Proceedings of the 1990 ACM/IEEE conference on Supercomputing (Supercomputing '90). IEEE Computer Society Press, Los Alamitos, CA, USA, 2-11.

[15] E. Polizzi, Density-Matrix-Based Algorithms for Solving Eigenvalue Problems, Phys. Rev. B. Vol. 79, 115112 (2009)

[16] E. Polizzi, A High-Performance Numerical Library for Solving Eigenvalue Problems: FEAST solver User's guide, arxiv.org/abs/1203.4031 (2012)