New Multithreaded Ordering and Coloring Algorithms for Multicore Architectures

Md. Mostofa Ali Patwary[1], Assefaw H. Gebremedhin[2], and Alex Pothen[3]

[1] University of Bergen, Norway. Mostofa.Patwary@ii.uib.no
[2] Purdue University, West Lafayette, IN. agebreme@purdue.edu
[3] Purdue University, West Lafayette, IN. apothen@purdue.edu

Abstract. We present new multithreaded vertex *ordering* and *distance-k graph coloring* algorithms that are well-suited for multicore platforms. The vertex ordering techniques rely on various notions of "degree", are known to be effective in reducing the number of colors used by a *greedy* coloring algorithm, and are generic enough to be applicable to contexts other than coloring. We employ *approximate degree* computation in the ordering algorithms and *speculation* and *iteration* in the coloring algorithms as our primary tools for breaking sequentiality and achieving effective parallelization. The algorithms have been implemented using OpenMP, and experiments conducted on Intel Nehalem and other multicore machines using various types of graphs attest that the algorithms provide scalable runtime performance. The number of colors the algorithms use is often close to optimal. The techniques used for computing the ordering and coloring in parallel are applicable to other problems where there is an inherent ordering to the computations that needs to be relaxed for increasing concurrency.

1 Introduction

Multicore platforms with support for multithreading have become commonplace and have reinvigorated the development of shared-memory parallel algorithms. We present new multithreaded algorithms well-suited for such platforms for two inter-related collection of graph problems: vertex *ordering* and *distance-k coloring*. Distance-1 coloring is used (among many others) in parallel scientific computing to discover tasks that can be carried out or data elements that can be updated concurrently [7, 8]. Distance-2 coloring is an archetypal model used in the efficient computation of sparse Jacobian and Hessian matrices [5]. We rely on *greedy* algorithms that incorporate a vertex *ordering* stage to solve the coloring problems. The vertex ordering techniques we consider are formulated in a manner independent of a coloring algorithm. They are known to be effective in reducing the number of colors used by a greedy coloring algorithm, but are of interest in their own right with applications in areas outside coloring.

The ordering and coloring algorithms we consider are challenging to parallelize as the computation involved is inherently sequential. We overcome this fundamental challenge using approaches that potentially are useful for other problems as well. For the ordering algorithms, we employ *approximate degree* computation as a mechanism for increasing concurrency. We show that such an approach leads to a scalable performance, whereas an approach that is faithful to the serial behavior of the ordering does not. The approximation-based method does not only lead to scalable performance, but is also far simpler. For the coloring algorithms, we use *speculation* and *iteration* as our primary

ingredients for achieving scalable performance. We focus in this work on distance-2 coloring, although the techniques are equally applicable to distance-1 coloring. The algorithms we have developed are implemented using OpenMP. Experiments conducted on an Intel Nehalem machine using a set of graphs designed to cover a wide spectrum of input types show scalable runtime performance. The number of colors the algorithms use is nearly the same as in the serial case, which in turn is often close to optimal.

Like many other graph algorithms, the algorithms we have considered are plagued by several performance impediments besides low concurrency: poor data locality, irregular memory access pattern, and high data access to computation ratio. Our primary focus in this work is on algorithmic techniques and we pay almost no attention to optimization techniques that could further enhance performance.

Preliminaries, Related Work, and Organization A distance-k coloring of a graph $G = (V, E)$ is an assignment of positive integers, called *colors*, to vertices such that any two vertices connected by a path consisting of at most k edges receive different colors. The objective in the distance-k coloring problem is to minimize the number of colors used, and the problem is known to be NP-hard for every fixed integer $k \geq 1$ (see [5] for pointers to references). Previous work has shown that a *greedy* coloring algorithm—an algorithm that visits vertices sequentially in some *order* in each step assigning a vertex the *smallest* permissible color—is quite effective in practice.

The order in which vertices are processed determines the number of colors used by the algorithm. In an earlier work [6], we identified three ordering techniques, called *Smallest Last* (SL), *Dynamic Largest First* (DLF), and *Incidence Degree* (ID) that are particularly effective in reducing the number of colors used by a greedy coloring algorithm and are generic enough to be useful in other contexts. In particular, the three ordering techniques are characterized (in [6]) purely in terms of *relative* vertex degrees, in a manner decoupled from the coloring algorithm that could use them. Such a characterization makes the orderings of interest in their own right and helps to more easily see their connections with other graph problems. For example, an SL ordering has interesting relationship with such graph concepts as degeneracy, core and arboricity (see [5] for some pointers). In this paper, we present algorithms—which are the first to the best of our knowledge—for *parallelizing* the aforementioned ordering techniques on multithreaded, shared-memory architectures. The algorithms are discussed in Sect. 2.

Using *speculation* and *iteration* as basic ingredients, a framework for effective parallelization of greedy distance-1 coloring on *distributed-memory* architectures was developed in [2]. The framework was extended to distance-2 coloring and related problems in [1]. Recently, a multithreaded algorithm derived from the framework in [2] and tailored for *shared-memory* architectures has been developed for the distance-1 coloring problem in [3]. We present in this paper a similar algorithm for distance-2 coloring on shared memory platforms. The algorithm is described in Sect. 3. We present experimental results in Sect. 4 and conclude in Sect. 5.

2 Vertex Ordering

2.1 The Serial Framework

We give in Algorithm 1 a succinct summary of a *template* for the ordering techniques SL, DLF and ID in the serial setting. Table 1 shows how the template is *specialized* in

Algorithm 1 Template for serial ordering (SL, DLF, ID). Input: graph $G = (V, E)$. Output: An ordered list W of the vertices in V. B is a two-dimensional array used for maintaining *unordered* vertices binned according to their "degrees".

```
1:  procedure OrderingTemplate(G = (V, E))
2:      for each vertex v ∈ V do
3:          init d(v)
4:          B[d(v)] ← B[d(v)] ∪ {v}
5:      init i                          ▷ i is position in W where next vertex in the order is placed
6:      while check i do                                          ▷ there remain vertices to order
7:          locate j*, an appropriate extreme index j where B[j] is non-empty
8:          Let v be a vertex drawn from B[j*]
9:          W[i] ← v
10:         B[j*] ← B[j*] \ {v}
11:         for each vertex w ∈ adj(v) such that w is in B do
12:             B[d(w)] ← B[d(w)] \ {w}
13:             update d(w)
14:             B[d(w)] ← B[d(w)] ∪ {w}
15:         update i
```

Table 1. Table accompanying the ordering template in Algorithm 1

	SL	DLF	ID
L 3: `init` $d(v)$	$d(v) \leftarrow d(v, G)$	$d(v) \leftarrow d(v, G)$	$d(v) \leftarrow 0$
L 5: `init` i	$i \leftarrow \lvert V\rvert - 1$	$i \leftarrow 0$	$i \leftarrow 0$
L 6 `check` i	$i \geq 0$	$i \leq \lvert V\rvert - 1$	$i \leq \lvert V\rvert - 1$
L 7: `locate` j^*	$j^* = \min_j\{B[j] \neq \emptyset\}$	$j^* = \max_j\{B[j] \neq \emptyset\}$	$j^* = \max_j\{B[j] \neq \emptyset\}$
L 13: `update` $d(w)$	$d(w) \leftarrow d(w) - 1$	$d(w) \leftarrow d(w) - 1$	$d(w) \leftarrow d(w) + 1$
L 15: `update` i	$i \leftarrow i - 1$	$i \leftarrow i + 1$	$i \leftarrow i + 1$

the three cases. The key idea in the definition (and computation) of these orderings is the use of a *dynamically* changing quantity, the *back* or *forward degree* of vertices. The back degree of a vertex v is the number of vertices that are adjacent to v in G and appear *before* v in the ordering, and the forward degree of v is the number of vertices that are adjacent to v in G and appear *after* v in the ordering. In Algorithm 1 and elsewhere in this paper, the dynamic degree (back or forward) of a vertex v is denoted by $d(v)$, and the *static* degree of the vertex in the input graph G is denoted by $d(v, G)$.

To arrive at an efficient implementation, a two-dimensional array B is used in Algorithm 1 to maintain vertices that are not yet ordered in *bins* according to their dynamic degrees. Specifically $B[j]$ stores a set of unordered vertices where each member vertex u has a current dynamic degree $d(u)$ equal to j. The output of Algorithm 1 is given by the ordered list W of the vertices where $W[i]$ stores the ith vertex in the ordering. In SL, the ordering W is computed right-to-left ($i = |V| - 1$ down to $i = 0$), whereas the ordering in DLF and ID is computed left-to-right ($i = 0$ up to $i = |V| - 1$). The ith vertex in SL ordering is a vertex with the *smallest* back degree among the vertices not yet ordered, in a DLF ordering it is a vertex with the *largest* forward degree among the vertices not yet ordered, and in an ID ordering it is a vertex with the *largest* back degree among the vertices not yet ordered. The rationale behind each of these ordering

techniques in the context of a coloring algorithm is to bring vertices that are likely to be highly constrained in choice of colors early in the ordering.

In Line 7 in Algorithm 1, we determine the ith vertex in the ordering in constant time by maintaing a pointer to the last element in the smallest (or largest) index j such that $B[j]$ is non-empty. Once the ith vertex v in the ordering is determined (and removed from B), each unordered vertex w adjacent to v is moved from its current bin in B to an appropriate new bin. With suitable pointer techniques the relocation can also be performed in constant time [6]. Thus the work involved in the ith step of Algorithm 1 is proportional to $d(v, G)$, and the overall complexity of the algorithm is $O(|E|)$.

We point out another interesting connection between the template in Algorithm 1 and an ordering used for an entirely different purpose: an ID ordering obtained by Algorithm 1, when reversed, corresponds to an ordering obtained by the *maximum cardinality search* algorithm [9], which arises in the context of solving sparse linear systems.

2.2 Parallel Ordering

We parallelized the three ordering techniques SL, DLF, and ID employing a common paradigm, but we restrict the presentation in this paper to only SL ordering.

We developed two different approaches for the parallelization. The first approach aims at parallelizing the ordering closely maintaining the serial behavior, while the second approach settles for an approximate solution in favor of increased concurrency. In both approaches, we assume p threads are available and utilized, and we denote by $t\,(v)$ the thread with which the vertex v is initially associated.

The First Approach—Regular Algorithm 2 outlines the first approach. The first task Algorithm 2 parallelizes is the population of the global bin array B. To achieve this, with each thread T_k, $1 \leq k \leq p$, a *local* two-dimensional array B_k is associated. The p local arrays are first populated in parallel (the for-loop in Lines 2–4). Then, the contents are gathered into the global array B, where the parallelization is now switched to run over bins, as shown in the for-loop in Lines 5–8. There and elsewhere in this paper, $\delta(G)$ and $\Delta(G)$ denote the minimum and maximum degree in G, respectively.

The remainder of Algorithm 2 mimics the serial algorithm (Algorithm 1). In the serial algorithm, in each step of the while loop, a *single* vertex—a vertex with the *smallest* current dynamic degree j^*—is ordered and its neighbors' locations updated in B. However, the bin $B[j^*]$ could contain *multiple* vertices. Algorithm 2 takes advantage of this opportunity and strives to order such vertices and update their neighborhoods in parallel. There are a few potential problems that need to be attended while doing so.

– *Problem*: A pair of vertices u and v in $B[j^*]$ are adjacent to each other. In such a case, a thread processing one of the vertices, say u, could try to move the vertex v to another bin while another thread at the same time attempts to order v, making the result inconsistent. *Solution*: While ordering the vertex u, we avoid updating the location of the vertex v in B, and instead order v as well in the current step (see Lines 12–17).

– *Problem*: Removal of multiple vertices from the same bin, say $B[j]$. Suppose two vertices u and v from $B[j^*]$ have a common neighbor w in $B[j]$. In the serial case, u and v would be ordered one after another, $d(w)$ would be reduced by 2, and w would be relocated twice. In the parallel case, two threads might try to remove w from $B[j]$ at the same time and the removal of w in constant time will make $B[j]$ inconsistent. Similarly, suppose two vertices u and v in $B[j^*]$ have respective neighbors w and x

Algorithm 2 A parallel SL ordering algorithm using p threads (the REGULAR variant). Input: graph $G = (V, E)$. Output: An ordered list W of the vertices in V. The array B is as in Algorithm 1, and the arrays B_t, R_t, and A_t are thread-private arrays; the latter two are used to remove or add vertices from or into the global array B.

```
1:  procedure SMALLESTLASTORDERING-REGULAR(G = (V, E))
2:      for each vertex v ∈ V in parallel do
3:          d(v) ← d(v, G)
4:          B_{t(v)} [d (v)] ← B_{t(v)} [d (v)] ∪ {v}
5:      for each bin j ∈ {δ (G) , . . . , Δ (G)} in parallel do
6:          for k = 1 to p do
7:              for each vertex v ∈ B_k [j] do
8:                  B [j] ← B [j] ∪ {v}                          ▷ note that j = d (v)
9:      i ← |V|
10:     while i ≥ 0 do
11:         Let j* denote the smallest index j such that B [j] is non-empty
12:         for each vertex v ∈ B [j*] in parallel do
13:             for each vertex w ∈ adj (v) such that w is in B do
14:                 if w ∉ R_{t(v)} then
15:                     R_{t(v)} [d (w)] ← R_{t(v)} [d (w)] ∪ {w}
16:                 r (w) ← r (w) + 1                            ▷ atomic operation
17:             W [i] ← v; i ← i − 1                             ▷ critical statements
18:         for each bin j ∈ {j*, . . . , Δ (G)} in parallel do
19:             for k = 1 to p do
20:                 for each vertex v ∈ R_k [j] do
21:                     if r (v) > 0 then
22:                         B [j] ← B [j] \{v}                   ▷ note that j = d (v)
23:                         d (v) ← d (v) − r (v); r (v) ← 0
24:                         A_{t(v)} [d (v)] ← A_{t(v)} [d (v)] ∪ {v}
25:         for each bin j ∈ {j*, . . . , Δ (G)} in parallel do
26:             for k = 1 to p do
27:                 for each vertex v ∈ A_k [j] do
28:                     B [j] ← B [j] ∪ {v}                      ▷ note that j = d (v)
```

such that $d(w) = d(x) = j$. In the parallel case, two threads might try to remove w and x from $B[j]$ at the same time while processing u and v in parallel and the removals of w and x in constant time will also make $B[j]$ inconsistent. *Solution*: We let each thread T_k, $1 \leq k \leq p$, maintain its own two-dimensional *removal* array R_k, where it stores vertices to be removed from B while the parallel ordering of $B[j^*]$ happens (see the for loop in Lines 13–16). The removal from B takes place once the ordering of vertices in $B[j^*]$ is completed. Since for any two bins $B[j]$ and $B[j']$ the removal from $B[j]$ is independent of the removal from $B[j']$, these could be done in parallel, as shown in Lines 18–24.

– *Problem*: Addition of multiple vertices to the same bin, say $B[j]$. *Solution*: We address this concern by using a similar technique as in the second bullet item. We let each thread

Algorithm 3 A parallel SL ordering algorithm on p threads (the RELAXED variant). Input: graph $G = (V, E)$. Output: An ordered list W of the vertices in V.

```
 1: procedure SMALLESTLASTORDERING-RELAXED(G = (V, E))
 2:     for each vertex v ∈ V in parallel do
 3:         d(v) ← d(v, G)
 4:         B_{t(v)}[d(v)] ← B_{t(v)}[d(v)] ∪ {v}
 5:     i ← |V|
 6:     for k = 1 to p in parallel do
 7:         while i ≥ 0 do
 8:             Let j* be the smallest index j such that B_k[j] is non-empty
 9:             Let v be a vertex drawn from B_k[j*]
10:             B_k[j*] ← B_k[j*] \ {v}
11:             for each vertex w ∈ adj(v) do
12:                 if w ∈ B_k then
13:                     B_k[d(w)] ← B_k[d(w)] \ {w}
14:                     d(w) ← d(w) − 1
15:                     B_k[d(w)] ← B_k[d(w)] ∪ {w}
16:             W[i] ← v; i ← i − 1                    ▷ critical statements
```

maintain its own two-dimensional *addition* array A_k. Again, the addition of vertices to different bins in B can be done in parallel, as shown in Lines 25–28.

The Second Approach—Relaxed Our second approach for parallelizing the SL ordering algorithm abandons the use of the global array B altogether, and works only with the local arrays B_k associated with each thread T_k. In updating locations of neighbors of a vertex, a thread T_k checks whether or not the vertex w desired to be relocated is in the thread's local array B_k. If w is indeed in B_k it is relocated by the same thread, if not, it is simply ignored. In this manner, only *approximate* dynamic degrees are used while computing the ordering. The approach is formalized in Algorithm 3.

3 Parallel Distance-2 Coloring

The sequential greedy distance-2 coloring algorithm we seek to parallelize iterates over the vertex set V of the graph G, in each step assigning a vertex v the smallest color not used by any of its distance-2 neighbors. It can be implemented such that its complexity is $O(|V| \cdot \overline{d}_2)$, where $\overline{d}_2$ denotes the average number of distinct paths of length at most two edges leaving a vertex [5]. Algorithm 4 shows how we have parallelized the greedy algorithm in this work. The algorithm has two phases, both of which are performed in parallel, and runs in an iterative fashion. In the first phase of each round of the iteration, threads concurrently color their respective vertices in a *speculative* manner (paying attention to already available color information). In this phase, two vertices that are distance-2 neighbors with each other and are handled by two different threads may be colored concurrently and receive the same color, causing a *conflict*. In the second phase, threads concurrently check the validity of colors assigned to their respective vertices in the current round and identify a set of vertices that needs to be re-colored in the next round to resolve any detected conflicts. The algorithm terminates when every vertex has been colored correctly. In the event of a conflict, it suffices to re-color one of the two

Algorithm 4 An iterative parallel algorithm for distance-2 coloring using p threads. Input: graph $G = (V, E)$. Output: a vertex-indexed array $color$ [] indicating colors of vertices. The vertex set V is assumed to be *ordered*.

```
1:  procedure ITERATIVED2COLORING(G = (V, E))
2:      U ← V
3:      while U ≠ ∅ do
4:          for each vertex v ∈ U in parallel do                ▷ Phase 1: tentative coloring
5:              for each vertex w ∈ adj (v) do
6:                  mark color [w] as forbidden to vertex v
7:                  for each vertex x ∈ adj (w) and x ≠ v do
8:                      mark color [x] as forbidden to vertex v
9:              Pick the smallest permissible color c for vertex v
10:         R ← ∅                                  ▷ R denotes the set of vertices to be recolored
11:         for each vertex v ∈ U in parallel do                ▷ Phase 2: conflict detection
12:             cont ← true
13:             for each vertex w ∈ adj (v) and cont = true do
14:                 if color [v] = color [w] and v > w then
15:                     R ← R ∪ {v}; break
16:                 for each vertex x ∈ adj (w) and v ≠ x do
17:                     if color [v] = color [x] and v > x then
18:                         R ← R ∪ {v}; cont ← false; break
19:         U ← R
```

involved vertices to resolve the conflict. In Algorithm 4 (see Lines 14 and 17), we used the value (id) of vertices to decide the vertex to re-color. Other strategies, such as the use of random numbers associated with vertices, are also possible [2].

Although the tentative coloring and conflict detection phases in each round iterate over the same set U of vertices performing similar operations per vertex visit, the runtime of the conflict detection phase can be significantly reduced by terminating the search for a conflict in the distance-2 neighborhood of a vertex v as soon as the first conflict impacting v is discovered. This is achieved using the **break** statements in Lines 15 and 18. Note that the *cont* boolean variable in Line 12 is used to break out of the for-loop in Line 13 due to a condition in the for-loop in Line 16. Thanks to the use of the early breaks, we observed that the conflict detection phase typically takes roughly around 25% of the overall runtime of the algorithm; without the breaks the conflict detection phase would have taken the same time as the tentative coloring phase.

Scheduling. In the parallel coloring algorithm we just described as well as the parallel ordering algorithms discussed in Sect. 2.2, the runtime performance of the algorithms depends on the manner in which vertices are scheduled on threads. In the results we report in the next section, we used the *dynamic* scheduling option of OpenMP.

4 Experimental Results

In this section we present results on experiments performed on an Intel Nehalem machine equipped with Intel(R) Core(TM) i7 CPU 860 processors running at 2.8GHz. The system has 4 cores with 2 threads on each. The total memory size is 16 GB, with 4×32

Table 2. Structural properties of the various graphs in the testbed: scientific computing (sc), rmat-random (er), rmat-good (g), and rmat-bad (b). Δ denotes maximum degree in G.

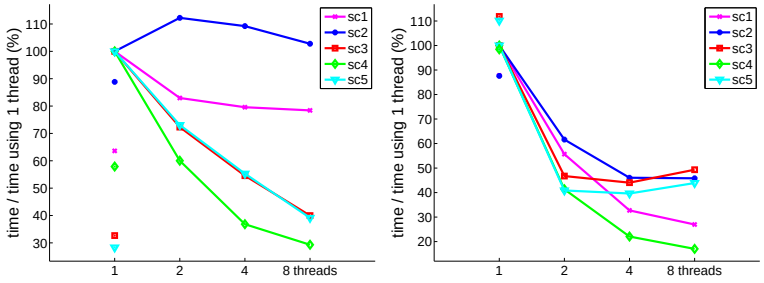
Name	$\|V\|$	$\|E\|$	Δ	Name	$\|V\|$	$\|E\|$	Δ
sc1 (bone010)	986,703	35,339,811	80	g1	262,144	2,093,552	558
sc2 (af_shell10)	1,508,065	25,582,130	34	g2	524,288	4,190,376	618
sc3 (nlpkkt120)	3,542,400	46,651,696	27	g3	1,048,576	8,382,821	802
sc4 (er1)	16,777,216	134,217,651	138	g4	2,097,152	16,767,728	1,069
sc5 (nlpkkt160)	8,345,600	110,586,256	27	g5	4,194,304	33,541,979	1,251
er1	262,144	2,097,104	98	b1	262,144	2,067,860	4,493
er2	524,288	4,194,254	94	b2	524,288	4,153,043	6,342
er3	1,048,576	8,388,540	97	b3	1,048,576	8,318,004	9,453
er4	2,097,152	16,777,139	102	b4	2,097,152	16,645,183	14,066
er5	4,194,304	33,554,349	109	b5	4,194,304	33,340,584	20,607

KB Instruction and 4×32 KB Data Level-1 cache, 4×256 KB Level-2 cache, and 8 MB shared Level-3 cache. The operating system is GNU/Linux.

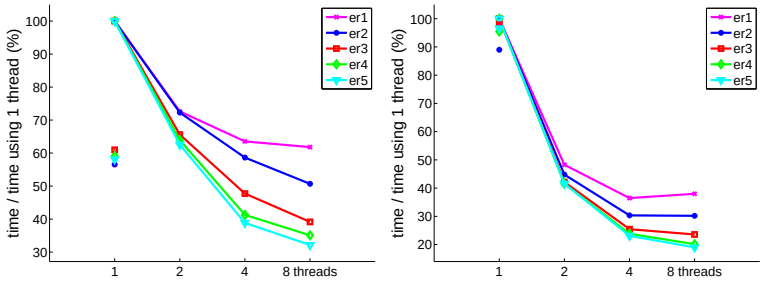
Our testbed consists of 20 graphs. Five of them are real-world graphs drawn from various *scientific computing* (sc) applications and are downloaded from the University of Florida Sparse Matrix Collection. The remaining 15 are synthetically generated using the R-MAT algorithm [4]. By combining the four input parameters of the R-MAT algorithm in various ways (the sum of the parameters needs to be equal to one), it is possible to generate graphs with varying properties. We generated three types of graphs:
(i) *Erdös-Renyi random* (er) graphs, using the set of parameters $(0.25, 0.25, 0.25, 0.25)$;
(ii) *small-world type 1* (g) graphs, using the set of parameters $(0.45, 0.15, 0.15, 0.25)$;
(iii) *small-world type 2* (b) graphs, using the set of parameters $(0.55, 0.15, 0.15, 0.15)$.
These three graph types vary widely in terms of *degree distribution* of vertices and *density of local subgraphs* and represent a wide spectrum of input types posing varying degrees of difficulty for the ordering and coloring algorithms. The er graphs have *normal* degree distribution, whereas the g (for "good") and b ("bad") graphs contain many dense local subgraphs (by good and bad is meant relatively "easy" and "hard" input types). The good and bad graphs differ primarily in the magnitude of maximum vertex degree they contain, the bad graphs have much larger maximum degree. Table 2 provides structural information on all 20 test graphs.

Figure 1 shows scalability results on the two parallel Smallest Last ordering algorithms, SL-Regular (Algorithm 2) and SL-Relaxed (Algorithm 3). The plots show runtimes for various numbers of threads *normalized* by the runtime when 1 thread is used. The raw runtime numbers for the 1 thread case along with the runtime of the pure *sequential* SL ordering and distance-2 coloring algorithms are provided in Table 3. Clearly, the algorithm SL-Regular scaled poorly especially for the sc and rmat-b graphs, whereas SL-Relaxed scaled well across all the graph types tested. We therefore present further results using the better performing algorithm SL-Relaxed.

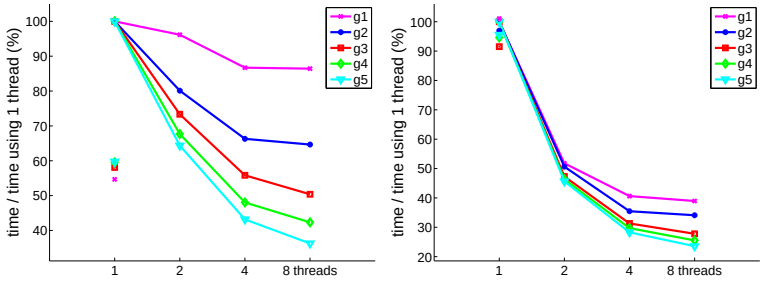
Figure 2 shows scalability results for the parallel distance-2 coloring algorithm (Algorithm 4) while using the SL-Relaxed algorithm for parallel ordering. The left column shows runtime results considering *only* the coloring stage, whereas the right column shows results on *total* (ordering plus coloring) time. Since distance-2 coloring takes substantially more time than the ordering (recall that the respective sequential com-



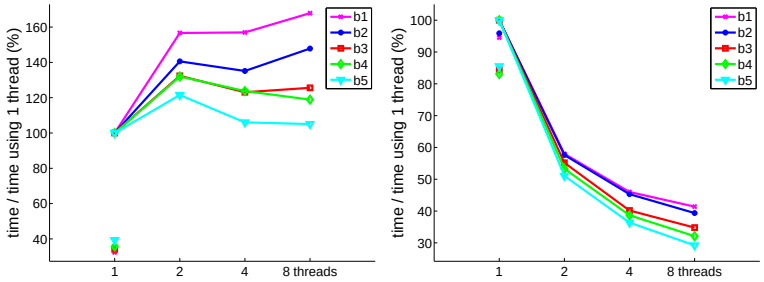
(a) Scientific computing (sc) graphs



(b) RMAT-ER (er) graphs



(c) RMAT-G (g) graphs



(d) RMAT-B (b) graphs

Fig. 1. Scalability results on the two parallel SL ordering algorithms. Left column: Algorithm 2 (SL-Regular). Right column: Algorithm 3 (SL-Relaxed). The plots show runtimes normalized by the runtime when 1 thread is used; the raw numbers for the case of 1 thread are listed in Table 3. Also shown are data points corresponding to runtime of the pure *sequential* algorithm normalized by the runtime of the parallel algorithm run on 1 thread.

Table 3. Runtime in seconds of the pure sequential algorithms, and of the parallel algorithms when run using one thread. OT shows ordering time, and CT shows distance-2 coloring time.

	SL-Seq.		SL-Relaxed		SL-Regular			SL-Seq.		S L-Relaxed		SL-Regular	
	OT	CT	OT	CT	OT	CT		OT	CT	OT	CT	OT	CT
sc1	1.11	20.66	1.18	30.45	1.73	31.18	g1	0.18	2.68	0.18	3.82	0.32	3.84
sc2	0.83	6.83	0.87	10.13	0.91	10.25	g2	0.45	6.31	0.42	8.86	0.74	9.03
sc3	2.05	11.38	1.64	16.45	6.39	28.89	g3	1.02	16.22	1.07	23.25	1.74	23.69
sc4	30.54	306.47	31.19	452.76	51.68	479.81	g4	2.49	43.16	2.54	61.98	4.18	65.64
sc5	5.15	27.51	4.29	39.86	17.68	73.91	g5	5.86	119.20	6.01	168.64	9.59	171.84
er1	0.18	1.45	0.18	2.13	0.32	2.21	b1	0.16	9.20	0.16	12.68	0.44	12.63
er2	0.43	3.30	0.45	5.02	0.71	5.23	b2	0.37	24.10	0.37	32.11	0.95	32.36
er3	1.22	9.24	1.20	12.75	1.84	13.54	b3	0.75	70.26	0.87	94.30	2.09	95.60
er4	2.77	22.48	2.86	33.62	4.54	36.07	b4	1.74	195.60	2.00	280.00	4.48	281.39
er5	6.30	57.13	6.43	83.74	10.51	88.77	b5	4.21	565.59	4.85	785.80	9.87	797.86

plexities are $O(|V| \cdot \overline{d}_2$ and $O(|V| \cdot \overline{d}_1))$, the scalability behavior of just the coloring stage is nearly identical to that of the overall execution. It can be seen that the coloring algorithm (including the ordering stage) scaled well across all the graphs in the testbed.

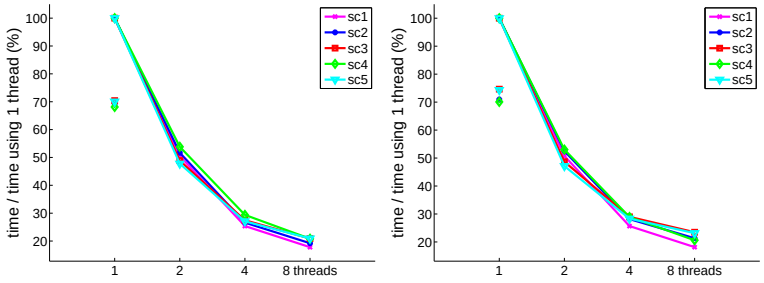
Also shown in Figures 1 and 2 is the runtime of a relevant *sequential* algorithm normalized by the runtime of the corresponding parallel algorithm run on 1 thread. This shows the performance advantage (besides functionality) gained by parallelization.

Figure 3 shows the number of colors the parallel distance-2 coloring algorithm (Algorithm 4) used while employing the SL-Relaxed ordering algorithm. In each subfigure, a bar corresponding to the maximum degree (Δ) in a graph, which is a lower bound on the optimal number of colors needed to distance-2 color a graph, is included. It can be seen that the number of colors the parallel algorithm used remained nearly constant as the number of threads is increased for all except the sc graphs. Further, it can be seen that the number in each case is either optimal or very close to optimal.

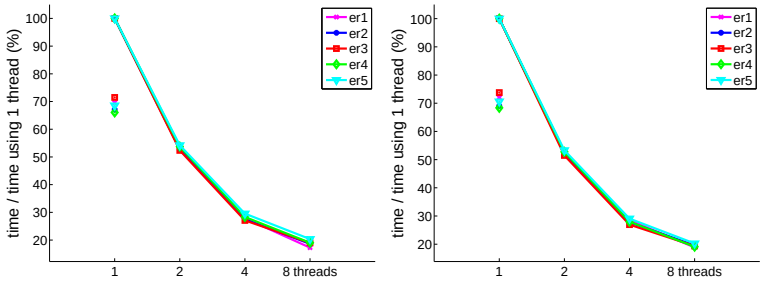
5 Conclusion and Future Work

We presented new parallel ordering and coloring algorithms and a small set of experimental results demonstrating scalable performance on a multicore machine supporting a modest number of threads. Some details and experimental results were omitted for space consideration. In future work, we intend to conduct further studies and provide more extensive results using machines supporting much larger number of threads. One issue we will investigate at large thread count is runtime scalability while maintaing quality of serial solution (to avoid increase in number of colors for some input types as those observed in Figure 3 a). The color choice strategy (see Line 9 of Algorithm 4) used in all of the results reported in this paper is *First Fit*, i.e., each thread searches for a permissible color for a vertex starting from *color 1*. We intend to investigate the merits of alternative color choice strategies (such as Staggered First Fit, Least Used, Random etc [2]) that could reduce the likelihood of conflicts.

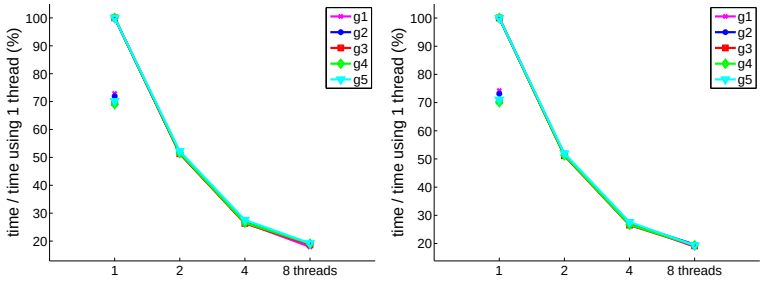
Acknowledgements We thank Fredrik Manne and Duc Nguyen for helpful discussions, and the referees for their helpful comments. One of the referees pointed out the relevance of reference [9]. This research was supported by the U.S. Department of Energy through the CSCPAES Institute grant DE-FC02-08ER25864 and by the National Science Foundation through grant CCF-0830645.



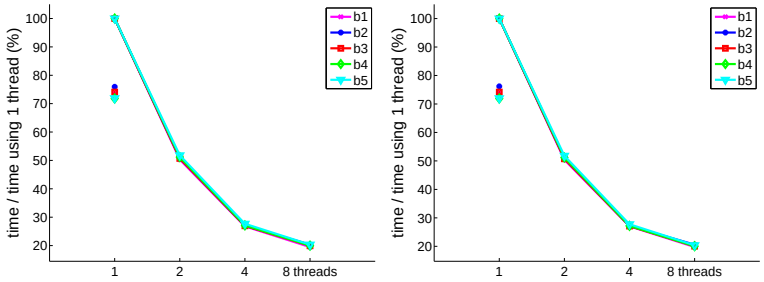
(a) Scientific computing (sc) graphs



(b) RMAT-ER (er) graphs

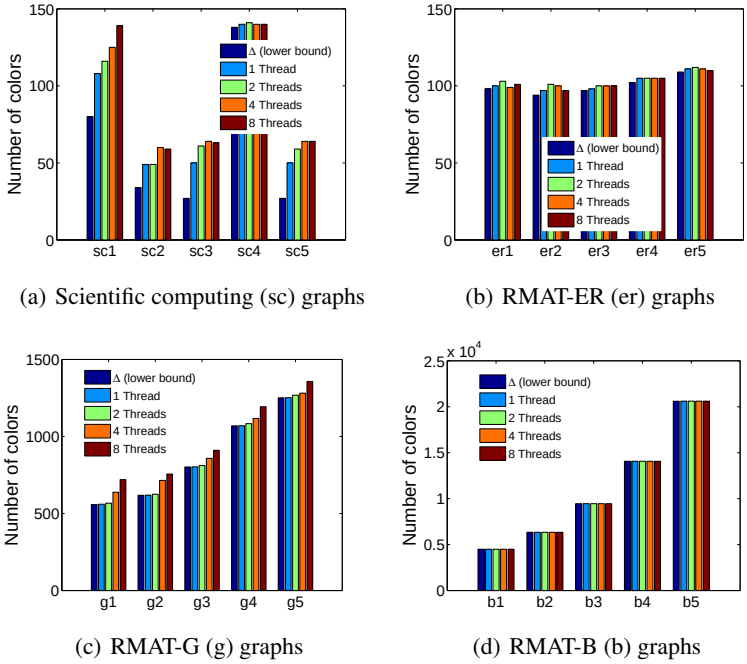


(c) RMAT-G (g) graphs



(d) RMAT-B (b) graphs

Fig. 2. Scalability results on the parallel distance-2 coloring algorithm (Algorithm 4) while employing the parallel ordering algorithm SL-Relaxed (Algorithm 3). Left column: only distance-2 coloring time. Right column: ordering plus distance-2 coloring time. The plots show runtimes normalized by the runtime when 1 thread is used; the raw numbers for the case of 1 thread are listed in Table 3. Also shown are data points corresponding to runtime of the pure *sequential* algorithm normalized by the runtime of the parallel algorithm run on 1 thread.



(a) Scientific computing (sc) graphs

(b) RMAT-ER (er) graphs

(c) RMAT-G (g) graphs

(d) RMAT-B (b) graphs

Fig. 3. Number of colors used by the parallel distance-2 coloring algorithm (Algorithm 4) while employing the SL-Relaxed ordering algorithm (Algorithm 3) for various thread counts. The first bar in each subfigure shows the lower bound Δ on the optimal number of colors.

References

1. D. Bozdağ, U.V. Catalyurek, A. H. Gebremedhin, F. Manne, E. G. Boman, and F. Ozgunner. Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation. *SIAM J. Sci.Comput.*, 32(4):2418–2446, 2010.
2. D. Bozdağ, A. H. Gebremedhin, F. Manne, E. G. Boman, and U. V. Catalyurek. A framework for scalable greedy coloring on distributed-memory parallel computers. *Journal of Parallel and Distributed Computing*, 68(4):515–535, 2008.
3. U. Catalyurek, J. Feo, A.H. Gebremedhin, M. Halappanavar, and A. Pothen. Multithreaded algorithms for graph coloring. Submitted for journal publication, 2011.
4. D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1):2, 2006.
5. A.H. Gebremedhin, F. Manne, and A. Pothen. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005.
6. A.H. Gebremedhin, D. Nguyen, M.M.A. Patwary, and A. Pothen. ColPack: Graph coloring software for derivative computation and beyond. Submitted for journal publication, 2010.
7. M.T. Jones and P.E. Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Computing*, 20(5):753–773, 1994.
8. Y. Saad. ILUM: A multi-elimination ILU preconditioner for general sparse matrices. *SIAM J. Sci. Comput.*, 17:830–847, 1996.
9. R.E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.