Paradigms for Effective Parallelization of Inherently Sequential Graph Algorithms on Multi-core Architectures

Assefaw H. Gebremedhin[*], Mostofa Patwary[†], and Fredrik Manne[‡]

September 30, 2020

Abstract

The chapter describes two algorithmic paradigms, dubbed SPECULATION AND ITERATION and APPROXIMATE UPDATE, for parallelizing greedy graph algorithms and vertex ordering algorithms, respectively, on multi-core architectures. The common challenge in these two classes of algorithms is that the computations involved are inherently sequential. The efficacy of the paradigms in overcoming this challenge is demonstrated via extensive experimental study on two representative algorithms from each class and two Intel multi-core systems. The algorithms studied are: (i) greedy algorithms for distance-k coloring (for $k = 1$ and $k = 2$) and (ii) algorithms for two degree-based vertex orderings. The experimental results show that the paradigms enable the design of scalable methods that to a large extent preserve the quality of solution obtained by the underlying serial algorithms.

Introduction

Greedy graph algorithms—where an optimization problem defined on a graph is solved by processing vertices (or edges) sequentially one at a time, at each step making the "best local" decision—occur frequently in computations. For some graph problems, Minimum Spanning Tree, for instance, a greedy algorithm is indeed the way to get an optimal solution. For NP-hard graph problems that occur as a part in a larger computation, greedy algorithms are often the methods-of-choice as they provide good approximate solutions at low, often linear, runtime. Further, greedy algorithms naturally fit in the framework of *streaming algorithms* (Alon et al., 1999), where input is fed one item at a time.

[*]School of Electrical Engineering and Computer Science, Washington State University. Email: assefaw.gebremedhin@wsu.edu

[†]NVIDIA, Applied Deep Learning Research Group.

[‡]Department of Informatics, University of Bergen.

In some greedy algorithms iterating over vertices, the *order* in which vertices are processed determines the quality of the solution obtained by the greedy algorithm. One may then need to find, for example, a *degree-based ordering*, where the vertices of a graph are ranked such that the vertex at each position is of maximum or minimum degree in a suitably defined induced subgraph. Degree-based ordering techniques may also be needed in their own right as a stand-alone procedure for an independent objective.

These two inter-related classes of algorithms, greedy algorithms and ordering procedures, have one common feature: the computations involved are *inherently sequential*. Existing parallel algorithm design techniques, such as divide-and-conquer, partitioning, pipelining, pointer-jumping, etc, that are commonly discussed in parallel computing books (Jájá, 1992; Grama et al., 2003; Kurzak et al., 2010) fall short as useful guidelines for effectively parallelizing such algorithms. The parallel algorithm developer's "design toolbox" thus needs to be augmented with new techniques, especially in the present era where parallel computing has established itself in the mainstream.

Contributions of This Chapter

This chapter contributes to this goal by focusing primarily on multi-core and multi-threaded architectures. Specifically, the chapter examines two design paradigms that turn out to be effective for parallelizing inherently sequential algorithms. The first paradigm, dubbed SPECULATION AND ITERATION, aims at parallelizing greedy algorithms. The second, named APPROXIMATE UPDATE, targets parallelization of ordering algorithms.

The key idea in SPECULATION AND ITERATION is to:

> *maximize concurrency by tentatively tolerating potential inconsistencies and then detecting and resolving eventual inconsistencies later, iteratively.*

For this approach to be successful (in leading to scalable methods), inconsistencies need to be relatively rare occurrences. We demonstrate that this is in fact the case for practical problems by applying the paradigm to parallelize greedy algorithms for *distance-k coloring* (for $k = 1$ and $k = 2$). We find, for instance, that the inconsistencies discovered in the very first iteration in the resultant parallel coloring algorithms run on moderate-scale computing environments typically involve less than one percent of the total number of vertices for large, sparse graphs. More generally, the number of inconsistencies will depend on the ratio between the number of vertices and threads and the density of the input graph.

The key idea in the APPROXIMATE UPDATE paradigm is to:

> *minimize synchronization cost by opting for concurrent data structure update with approximate data instead of serialized data structure update with exact data.*

Obviously, the solution output by a parallel algorithm designed with this paradigm is *not* guaranteed to be the same as a solution obtained by a sequential algorithm. This is not a major concern, however, since in most computations needing ordering, a slight deviation from the optimal (serial) ordering is not only tolerable but a welcome tradeoff to enable parallelization. We consider in this work the parallelization of two vertex ordering types, known as *Smallest Last* (SL) (Matula, 1968) and *Incidence Degree* (ID) (Coleman and More, 1983), as representatives of our second target class of algorithms. For each of these ordering variants, we study an *approximate degree update* approach for parallelization. We show that the approach gives a scalable method that does not incur too much loss in quality of solution relative to a serial algorithm whereas a method that insists on exact degree update does not scale.

SL and ID ordering and greedy coloring algorithms are closely related: the orderings can be used in an initialization step of the coloring algorithm to reduce the number of colors used. However, SL and ID orderings are also of independent interest because of their use in areas outside coloring, including network analysis and linear solvers.

As platforms for evaluating the scalability of the parallel coloring and ordering algorithms designed using the proposed paradigms, we experiment with two moderate-size (desk-side or desk-top) multi-core systems based on Intel processors. We show that our algorithms generally scale well on both platforms, with varying performance on each.

The remainder of this chapter is organized around the two paradigms. First, this introductory section is wrapped up with a brief review of related work. The second section discusses Speculation and Iteration and the associated coloring problems, and the third section treats Approximate Update and the associated ordering problems. The datasets and computing platforms used in the experiments (common to both paradigms) are discussed in the second section.

Related Work

The Speculation and Iteration design *paradigm* is an outgrowth of a series of previous works in which the focus was the design of parallel algorithms for *specific* graph problems. The basic idea of using speculation for parallelizing greedy graph coloring algorithms was first introduced in Gebremedhin and Manne (2000). There it was used in the context of shared-memory parallelization of coloring algorithms, albeit without iteration. Instead, the conflict resolution phase was carried out just once, serially on one processor. The idea was later enhanced with randomization in Gebremedhin et al. (2002).

Speculation together with iteration formed the core of the framework for parallelizing distance-1 coloring on *distributed-memory* architectures developed in Bozdağ et al. (2008). The framework addressed a variety of additional performance requirements entailed by a distributed-memory setting: the input graph needs to be partitioned in a manner that minimizes communication cost; the speculative coloring phase performs better when organized in a coarse-grained

fashion with infrequent communications; the coloring of interior and boundary vertices needs to be scheduled carefully; etc. The framework was later extended to distance-2 coloring, where mechanisms for minimizing inter-processor communication in conflict detection and resolution are even more critical (Bozdağ et al., 2010).

A *multi-threaded* algorithm for distance-1 coloring derived from the framework of Bozdağ et al. (2008) and adapted for shared-memory multi-core architectures has been studied in Catalyurek et al. (2012). In the same work, the architecture-portable, speculation-based multi-threaded algorithm is contrasted with a dataflow-based multi-threaded algorithm custom-designed for the Cray XMT.

Several other recent research activities have successfully used speculation ideas for parallelization (Patwary et al., 2012; Sariyuce et al., 2011, 2012). Initial work on one of the approximate degree updates methods discussed here was presented in Patwary et al. (2011). Although developed in an entirely different context, ideas behind distributed auction algorithms (Zavlanos et al., 2008), broadly interpreted, bear some resemblance to the speculation paradigm discussed here. In yet another different context, the term speculation (or optimistic parallelization) is also used to refer to compiler and/or runtime techniques for automatic parallelization of serial codes (Pingali et al., 2011; Tian et al., 2009).

Speculation and Iteration

We begin this section with an abstract presentation of the Speculation And Iteration parallelization paradigm. We then illustrate its use by applying it to parallelize greedy algorithms for graph coloring problems.

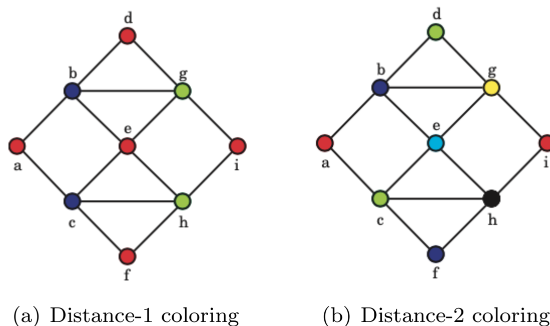
Generic Formulation

Suppose the input graph is $G = (V, E)$, the problem of interest involves operations on vertices, and there are p processing units, where $p \ll |V|$. We can formulate the Speculation and Iteration design technique in a generic fashion as shown in Algorithm 1. There, U denotes the set of "active" vertices.

The approach outlined in Algorithm 1 presupposes that resolving an inconsistency can be achieved via *local* re-evaluations. Intuitively, its viability is directly related to the rate at which the size of U drops from one iteration to the next—the faster the rate, the more viable the approach is. In other words, the approach is effective when the size of eventual inconsistencies discovered in an iteration is relatively small. We showcase the efficacy of this approach using greedy algorithms for *distance-k coloring*, for the cases $k = 1$ and $k = 2$. We proceed by first reviewing the underlying serial greedy coloring algorithms. We then discuss their parallelizations and show performance results. The dataset and platforms used for performance evaluation are discussed prior to the presentation of the performance results.

ALGORITHM 1: Generic formulation of the SPECULATION AND ITERATION parallelization paradigm.

Input: Graph $G = (V, E)$ and p processing units
1 $U \leftarrow V$ (U is the set of active vertices);
2 **while** U *is non-empty* **do**
3 Partition U in to p nearly equal subsets $U_1, \ldots, U_p$, and assign each subset to a distinct processing unit;
4 Solve the p sub-problems defined by the p subsets in parallel making speculative decisions as needed;
5 Check the validity of the p sub-solutions in parallel registering inconsistencies;
6 Reset U such that it contains only elements needing resolution;
7 **end**



(a) Distance-1 coloring (b) Distance-2 coloring

Figure 1: Illustration of a distance-1 and a distance-2 coloring of a graph on nine vertices. The distance-1 coloring example uses three colors with color classes $\{a, d, e, f, i\}$; $\{b, c\}$; and $\{g, h\}$. The distance-2 coloring uses six colors with colors classes $\{a, i\}$; $\{b, f\}$; $\{c, d\}$; $\{e\}$; $\{g\}$; and $\{h\}$.

Serial Coloring

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. See Figure 1 for an illustration of distance-1 and distance-2 coloring. The objective in the distance-k coloring problem is to minimize the number of colors used. The problem is NP-hard for every fixed integer $k \geq 1$ (Lin and Skiena, 1995).

Algorithms Despite known hardness (including in-approximability) results on distance-1 coloring, previous work has shown that a *greedy* 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 (Coleman and More, 1983). We review in Algorithm 2 an *efficient* formulation of the

ALGORITHM 2: A greedy distance-k coloring algorithm. The color-indexed array forbiddenColors is used to mark impermissible colors to a vertex.

```
  Input: Graph G = (V, E)
  Output: an array color[v] denoting colors assigned to vertices
1 Initialize forbiddenColors with some value a ∉ V;
2 for each v ∈ V do
3 |   for each w ∈ N_k(v) do
4 |   |   forbiddenColors[color[w]] ← v;
5 |   end
6 |   c ← min{i ≥ 1 : forbiddenColors[i] ≠ v};
7 |   color[v] ← c;
8 end
```

greedy distance-k coloring algorithm.

In Algorithm 2, and elsewhere in this chapter, $N_k(v)$ denotes the set of distance-k neighbors of the vertex v. The data structure color is a vertex-indexed array that stores the color of each vertex. The color-indexed array forbiddenColors is used to mark colors that are impermissible to a vertex v in a given step of the outer for-loop over vertices. In doing so, the vertex v itself is used as a 'stamp' thereby avoiding the need for re-initialization of forbiddenColors in a later step in which another vertex is colored. By the end of the inner for-loop of Algorithm 2, all of the colors that are impermissible to the vertex v are recorded in forbiddenColors. In Line 6, the array is scanned from left to right in search of the *lowest* positive index i at which a value different from v is encountered. The index i corresponds to the smallest permissible color c to the vertex v—and is thus assigned to v in Line 7.

The work done in populating the array forbiddenColors is proportional to $d_k(v)$, where $d_k(v)$, denoting "degree-k", is the number of edges in the graph induced by the vertices in $N_k(v) \cup \{v\}$. The search for the smallest allowable color c (Line 6) terminates after at most $|N_k(v)| + 1$ attempts, since the worst possible scenario is when each of the $|N_k(v)|$ neighbors of v uses a distinct color in the sequence $\{1, 2, \ldots, |N_k(v)|\}$; otherwise the sequence would contain a permissible color for v, allowing earlier termination. Thus the time complexity of Algorithm 2 is $O(|V| \cdot \overline{d}_k)$, where $\overline{d}_k$ is the average degree-k in the graph. This reduces to $O(|V| \cdot \overline{d}_1) = O(|E|)$ for distance-1 coloring. For distance-2 coloring, the expression can be bounded by $O(|E| \cdot \Delta)$, since $\overline{d}_2$ can be bounded by $\overline{d}_1 \cdot \Delta$, where Δ is the maximum degree in the graph.

Bounds on Number of Colors We quickly review obvious lower bounds on distance-1 and distance-2 coloring as well as upper bounds on the solution obtained by Algorithm 2 for the two coloring cases. The size of the largest induced clique in G, the clique number ω of G, is clearly a lower bound on the optimal number of colors needed for distance-1 coloring of G. Algorithm 2 in the distance-1 coloring case uses at most $\Delta + 1$ colors, where again Δ is the

maximum degree in G. The quantity Δ is a lower bound on the optimal number of colors needed to distance-2 color G. Algorithm 2 in the distance-2 coloring case uses at most $\min\{\Delta^2 + 1, |V|\}$ colors. The stated lower and upper bounds on distance-2 coloring imply that Algorithm 2 in the distance-2 coloring is an $O(\sqrt{|V|})$-approximation algorithm (McCormick, 1983).

Ordering The order in which vertices are processed in Algorithm 2 (Line 2) determines the number of colors used by the algorithm. Two ordering techniques known to be particularly effective in reducing number of colors—to values significantly lower than the bound $\Delta + 1$ in the case of distance-1 coloring—are *Smallest Last* (Matula, 1968; Matula et al., 1972) and *Incidence Degree* (Coleman and More, 1983). We use these two ordering techniques in our study of parallel coloring algorithms in this chapter. We defer a discussion of the details of the ordering techniques and their effective parallelization to the Approximate Update section.

Parallelization

We set out to parallelize Algorithm 2 using the SPECULATION AND ITERATION scheme we outlined in Algorithm 1. Let $G = (V, E)$ be the input graph, and p denote the number of available threads (processing units). We partition the vertex set V equally among the p threads. To ensure reasonable load balance, we assume that G is of bounded maximum degree. Our goal is to parallelize Algorithm 2 such that its complexity becomes $O(T_s(|G|)/p)$, where $T_s(|G|)$ is the runtime of the underlying serial algorithm. Algorithm 3 summarizes the key steps of the parallelized version.

The algorithm runs in rounds in an iterative fashion. Each round has two phases each of which is performed in parallel. In the first phase in each round, the current set of vertices to be colored (U) is equally divided among available threads. The threads then concurrently color their respective vertices in a *speculative* manner, paying attention to already available color information. In this phase, two vertices that are distance-k 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 only one of the two involved vertices to resolve the conflict. The function $r(\cdot)$ in Line 12 of Algorithm 3 is used to decide which of the two vertices to re-color. There are several choices for the function $r(\cdot)$: one can use, for example, vertex IDs or random numbers associated with each vertex.

On a PRAM (Parallel Random Access Machine (Jájá, 1992)) model, the parallel runtime in each round of Algorithm 3 is bounded by $O(|U| \cdot \overline{d}_k/p)$, assuming the input graph is of bounded maximum degree. Thus, provided

ALGORITHM 3: A Speculation and Iteration parallel algorithm for distance-k coloring. The array forbiddenColors is *private* to each thread.

```
   Input: Graph G = (V, E)
   Output: An array color indicating colors of vertices
 1 U ← V;
 2 while U ≠ ∅ do
 3 |   for each vertex v ∈ U in parallel do
 4 |   |   for each vertex w ∈ N_k(v) do
 5 |   |   |   forbiddenColors[color[w]] ← v;
 6 |   |   end
 7 |   |   c ← min{i ≥ 1 : forbiddenColors[i] ≠ v};
 8 |   |   color[v] ← c ;
 9 |   end
10 |   R ← ∅    (synchronization);
11 |   for each vertex v ∈ U in parallel do
12 |   |   if there exists a vertex w in N_k(v) such that color[v] = color[w]
   |   |    and r(v) > r(w) then
13 |   |   |   R ← R ∪ {v}    (critical);
14 |   |   |   Stop search in N_k(v);
15 |   |   end
16 |   end
17 |   U ← R   (synchronization);
18 end
```

that the algorithm terminates after a constant number of iterations, the overall complexity of the parallel algorithm is $O(|V| \cdot \overline{d}_k/p)$.

Setup for Performance Analysis

Implementation We implemented Algorithm 3, and also the parallel ordering algorithms to be discussed in Section 18, in C++ using *OpenMP*. The algorithms could, however, be implemented in *any* other programming model allowing threading. The performance of the resultant implementations depends on the manner in which tasks (a task in this case is work associated with a vertex) are *scheduled* on threads. OpenMP provides various scheduling options (static, dynamic, guided, runtime). In the results we report in this section and elsewhere in this chapter we use the option *dynamic* since it gave the best performance for a majority of our test cases.

Test platforms We use as our test platforms two moderate-size multi-core systems based on Intel processors. Table 1 gives an overview of the basic architectural features of the platforms along with information on the compilers we used. In all cases, the codes are compiled with -O3 optimization level. To further

Table 1: Summary of architectural features of the two Intel platforms used in our experiments.

	Xeon E7-4850 "Westmere-EX" (Nehalem-based Xeon)	**Core i7-860** "Lynnfield" (Nehalem microarch.)
Clock speed	2.0 GHz	2.8 GHz
# of sockets	4	1
Cores/socket	10	4
Threads/socket	20	8
Total cores	40	4
Memory	132 GB	16 GB
L3 cache, shared	20 MB	8 MB
L2 cache/core	256 KB	256 KB
Socket type	LGA[a] 1567	LGA 1156
Release date	2011	2009
Compiler (GNU, g++)	v 4.8.0	v 4.5.4

S ource: See `http://ark.intel.com` for further information

Notes: [a]Land Grid Array

improve performance in running the codes, we also use compiler-provided environment variables for realizing thread affinity (kmp_affinity) and mechanisms for realizing non-uniform memory access (numaactl).

Table 2: Structural properties of the graphs in the testbed.

	$\|V\|$	$\|E\|$	Δ	ω		$\|V\|$	$\|E\|$	Δ	ω		$\|V\|$	$\|E\|$	Δ	ω
`er1`	262K	2,097K	98	3	`g1`	262K	2,094K	558	6	`b1`	262K	2,068K	4,493	35
`er2`	524K	4,194K	94	3	`g2`	524K	4,190K	618	6	`b2`	524K	4,153K	6,342	39
`er3`	1,049K	8,389K	97	3	`g3`	1,049K	8,383K	802	6	`b3`	1,049K	8,318K	9,453	43
`er4`	2,097K	16,777K	102	3	`g4`	2,097K	16,768K	1,069	6	`b4`	2,097K	16,645K	14,066	$\geq$ 51
`er5`	4,194K	33,554K	109	3	`g5`	4,194K	33,542K	1,251	6	`b5`	4,194K	33,341K	20,607	$\geq$ 58

Dataset Our dataset consists of graphs generated using the R-MAT model (Chakrabarti and Faloutsos, 2006). We generated three types of graphs, named `er`, `g` and `b`, using the following R-MAT parameters:

`er`: $(0.25, 0.25, 0.25, 0.25)$

`g`: $(0.45, 0.15, 0.15, 0.25)$

`b`: $(0.55, 0.15, 0.15, 0.15)$

These three graph types vary widely in terms of *degree distribution* of vertices and *density of local subgraphs*. Therefore, they represent a wide spectrum of input types posing varying degrees of difficulty for the coloring and ordering

Table 3: Number of iterations and number of vertices colored in each iteration of the distance-1 coloring and distance-2 coloring versions of Algorithm 3 for select runs on the Xeon E5.

		D1 color.		D2 color.	
	# threads	# rounds	$\|U\|$ in each round	# rounds	$\|U\|$ in each round
er5	16	2	4,194K → 17	2	4,194K → 394
	24	2	4,194K → 19	2	4,194K → 795
	32	2	4,194K → 44	3	4,194K → 938 → 1
g5	16	2	4,194K → 136	8	4,194K → 2,164 → 151 → 48 → . . . 3 → 1
	24	2	4,194K → 119	9	4,194K → 2,209 → 344 → 105 → . . . 4 → 2
	32	2	4,194K → 153	10	4,194K → 2,458 → 536 → 187 → . . . 3 → 1
b5	16	3	4,194K → 209 → 2	20	4,194K → 11,525 → 2,086 → 1,046 → . . . 2 → 1
	24	2	4,194K → 515	26	4,194K → 16,217 → 3,503 → 1,908 → . . . 3 → 1
	32	3	4,194K → 377 → 1	24	4,194K → 20,747 → 5,240 → 2,438 → . . . 2 → 1

algorithms we consider. The `er` graphs (for *Erdös-Renyi* random graphs) have *normal* degree distribution. The `g` and `b` graphs in contrast have skewed-degree distributions and contain many more dense local subgraphs than the `er` graphs. The `g` and `b` graphs differ primarily in the magnitude of maximum vertex degree they contain—the `b` graphs have much larger maximum degree (see Catalyurek et al. (2012) for an analysis of the structures of similarly generated RMAT graphs).

Table 2 lists the number of vertices, the number of edges, the maximum degree Δ and the clique number ω in each graph in the testbed. Computing the clique number of a graph is an NP-hard problem. We calculated the clique numbers in Table 2 using fast maximum clique algorithms (exact and heuristic) we implemented (Pattabiraman et al., 2013). For the two graphs `b4` and `b5`, since the execution time of the exact, maximum clique algorithm was high, we settled for a solution provided by the heuristic, which finds a large, but not necessarily a largest clique in a graph. The listed numbers there, 51 and 58, are therefore lower bounds on the clique numbers.

Performance Results

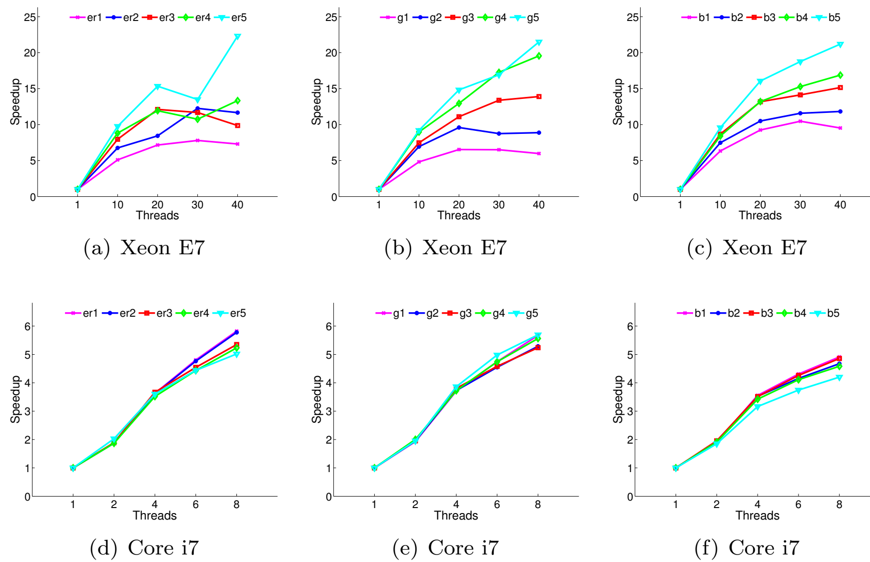
Scalability

As discussed in Section 8, the scheme outlined in Algorithm 3 would scale if the number of *iterations* the algorithm needs to terminate is relatively small. We find that number to be very small indeed in the extensive experiments we carried out with different types of input graphs and levels of concurrency. For the distance-1 coloring version of Algorithm 3, in particular, the number of iterations needed was typically found to be just two or three. Understandably, the distance-2 coloring version needed more iterations to terminate, but still the algorithm typically terminated within at most about two dozen iterations when the highest number of threads are employed.

Furthermore, especially in the distance-1 coloring case, we observed that

typically more than 99.9% of the vertices get their final colors in the *first* round! Put in another way, the number of conflicts that arise in the first round was found to be typically less than 0.1%. In the subsequent iterations, the size of conflicts, or the size $|U|$ of the vertices to be recolored, dropped dramatically from one iteration to the next. As an example of these observations, we give in Table 3 the number of iterations and the size of the set U in each iteration in the parallel distance-1 coloring and parallel distance-2 coloring algorithms for three of the largest graphs and select runs on the Xeon E5 machine.

The magnitudes of the number of iterations in Table 3 in general suggest that the Speculation and Iteration approach for parallelizing coloring algorithms is a viable framework. Looking at the numbers, it can be envisioned that best performance in terms of speedup might be attained if one were to stop the iteration and switch to sequential treatment of U once a predefined cutoff value for $|U|$ has been attained. We do not pursue this line of thought here. We work instead with the basic variant of Algorithm 3, where every iteration is performed in parallel.



(a) Xeon E7 (b) Xeon E7 (c) Xeon E7

(d) Core i7 (e) Core i7 (f) Core i7

Figure 2: Speedup of **distance-2 coloring** on the two platforms and three classes of graphs `er`, `g` and `b`.

Figure 2 shows speedup plots we obtain for the parallel distance-2 coloring algorithm for experiments run on the two test platforms and using all of the graphs in the dataset. In Figure 3 we show analogous results for the distance-1 coloring algorithm, but only for the `er` graphs in the dataset. We used in these experiments an approximate SL ordering (which we will discuss in Section 18) to reduce the number of colors used. In the plots in figures 2 and 3, the runtimes of the ordering step are, however, excluded for clarity of presentation.

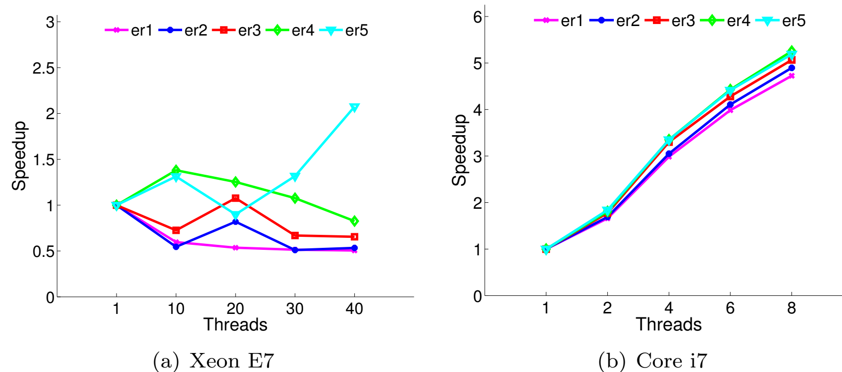
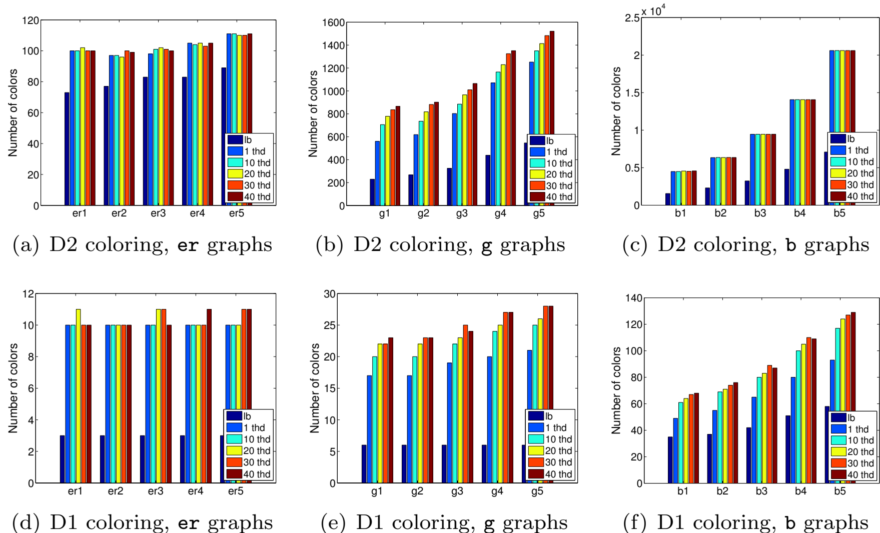


Figure 3: Speedup of **distance-1 coloring** on the two platforms and the `er` class of graphs.

From Figure 2 it can be seen that the distance-2 coloring algorithm scales well across both platforms and all graphs in the testbed. Further, it can be seen from Figure 3 that the scalability in the distance-1 coloring case is poorer than in the distance-2 coloring case, even though the number of iterations the distance-1 coloring algorithm needed was much smaller than what the distance-2 coloring algorithm needed. This is because the work involved in distance-1 coloring (which is $O(|E|)$) is substantially less than that in distance-2 coloring (which is $O(|E| \cdot \Delta)$), and hence the algorithm is more sensitive to memory performance.

We point out one difference we observe concerning performance on the Xeon E7 machine compared to performance on the Core i7 machine. In the speedup plots for the Xeon E7 machine in figures 2 and 3 (and elsewhere in this chapter), we report results for up to 40 threads, which amounts to using one thread per core out of the two available via hyperthreading. We do so because we did not observe any significant further reduction in runtime in going beyond 40 threads. In contrast, for the Core i7 machines, we report results for up to the maximum possible threads, which is 8. This amounts to taking advantage of hyperthreading. As can be seen from the results in figures 2 and 3, some performance gain can be achieved by doing so on these machines. It should be noted here that the *ideal* speedup expected in using two (hyper)threads on a single processor is considerably less than two—it is observed to be at most about 1.5 for most Intel architectures (Barker et al., 2008). In light of this, the decrease in slope of the speedup plots we see in figures 2 and 3 for the segments beyond 4 threads on the Core i7 is in agreement with one's expectation.

In the plots in figures 2 and 3 (and in similar plots in Section 18), the speedups are calculated by normalizing runtimes by the execution time of the relevant parallel algorithm run on *one* thread. This normalizing quantity is not the same as the pure *sequential* algorithm's runtime. In Table 4 we list raw execution times in seconds, on the Xeon E7 machine, of a parallel algorithm



(a) D2 coloring, `er` graphs (b) D2 coloring, `g` graphs (c) D2 coloring, `b` graphs

(d) D1 coloring, `er` graphs (e) D1 coloring, `g` graphs (f) D1 coloring, `b` graphs

Figure 4: Number of colors used in distance-2 coloring (top) and distance-1 coloring (bottom) while using an approximate SL ordering. The tests are run on the Xeon E7 machine with varying number of threads (1, 10, 20, 30, 40). The lowest bar in each subfigure, labeled *lb*, shows the lower bounds on the number of colors: Δ in the distance-2 coloring case and ω in the distance-1 coloring case.

run on one thread and of the corresponding sequential algorithm for a number of different algorithms of interest in this chapter. The results relevant for our discussion here are those of the distance-1 coloring and distance-2 coloring algorithms. These are indicated in the table by boldface fonts. We will return to the rest of the data in the table in the Approximate Update section.

Quality of Solution

We have so far presented experimental results on runtime and speedup. We now turn to the quality of the solution obtained by the parallel algorithms.

Previous research has established that the serial greedy distance-k coloring algorithm (Algorithm 2), when employing ordering techniques such as SL, gives near optimal solution on most practically relevant classes of graphs. We find that its parallelization using the speculation paradigm results in runtime speedup without compromising the quality of the solution obtained by the serial algorithm. Figure 4 supports this claim.

The upper row of the figure shows the number of colors the parallel distance-2 coloring version of Algorithm 3 uses while employing an approximate SL ordering for runs conducted on the Xeon E7. In each subfigure, six bars are shown for each graph. The shortest bar shows the maximum degree (Δ) in a graph, which is a lower bound on the optimal number of colors needed to distance-2

Table 4: Runtime in seconds of the pure sequential algorithms, and of the parallel algorithms when run using one thread, on the **Xeon E7** machine. For the SL ordering algorithm, two parallelizations are considered: Regular (Reg) and Relaxed (Rel).

	Sequential			Parallel, 1 thread SL		Parallel, 1 thread coloring	
	SL	**D1**	**D2**	Reg	Rel	**D1**	**D2**
`er1`	0.21	**0.06**	**1.17**	0.33	0.21	**0.11**	**1.66**
`er2`	0.52	**0.17**	**3.12**	0.77	0.50	**0.31**	**4.72**
`er3`	1.26	**0.49**	**9.28**	1.89	1.23	**0.89**	**14.78**
`er4`	3.46	**1.28**	**23.88**	5.30	3.78	**2.36**	**41.09**
`er5`	10.49	**3.68**	**66.00**	16.45	10.74	**6.44**	**101.27**
`g1`	0.21	**0.06**	**2.20**	0.36	0.20	**0.10**	**2.88**
`g2`	0.49	**0.14**	**5.37**	0.81	0.47	**0.27**	**7.63**
`g3`	1.20	**0.42**	**17.17**	1.89	1.16	**0.74**	**25.32**
`g4`	3.06	**1.13**	**46.96**	4.82	3.17	**2.05**	**70.34**
`g5`	8.24	**3.07**	**123.99**	12.46	8.56	**5.57**	**190.39**
`b1`	0.18	**0.05**	**7.89**	0.56	0.18	**0.08**	**9.46**
`b2`	0.43	**0.11**	**20.19**	1.20	0.43	**0.21**	**25.68**
`b3`	0.87	**0.31**	**69.81**	2.50	0.98	**0.55**	**93.78**
`b4`	2.12	**0.82**	**211.64**	5.59	2.40	**1.50**	**291.16**
`b5`	5.13	**2.05**	**606.23**	13.26	5.89	**3.78**	**860.43**

color a graph. The remaining five bars correspond to the number of colors used by the parallel algorithm when run using five different numbers of threads: 1, 10, 20, 30 and 40. The lower row of Figure 4 shows entirely analogous results for the parallel distance-1 coloring algorithm. In each subfigure there, the shortest of the six bars shows the clique number (ω) in a graph, which is a lower bound on the optimal number of colors needed to distance-1 color a graph.

We point out two observations from these results. First, it can be seen that the number of colors the parallel algorithm uses remains nearly constant as the number of threads is increased. This is true for both the distance-2 coloring and the distance-1 coloring algorithms. Note here that the number of colors the parallel algorithm uses when one thread is employed (bars labeled *1 thd* in the figures) is the same as the number of colors the *serial* algorithm would have used. Let us call this number C_{1thd}.

Second, it can be seen that the number C_{1thd} is fairly close to the *lower bound* on the optimal solution (bars labeled *lb* in the figures). Since a gap is expected to exist between *the lower bound on the optimal solution* and *the optimal solution*, C_{1thd} would actually be even closer to the optimal solution. Proceeding with comparison against the lower bound nonetheless, we observe that these algorithms do offer low approximation ratios. In the distance-1 coloring case, for instance, the number of colors in each subfigure is observed to be just a small constant γ times the lower bound ω. In the worst cases, γ is observed to be about 3 for the `er` graphs, 4 for the `g` graphs, and 2 for the `b` graphs.

Approximate Update

As mentioned in the Introduction, we use algorithms for obtaining Smallest Last and Incidence Degree orderings as examples to illustrate our second parallelization paradigm, APPROXIMATE UPDATE. We begin by reviewing the properties of these orderings and their serial algorithms. Then we discuss their parallelization and present performance results.

Degree-based Vertex Orderings

We define SL and ID ordering in terms of a dynamic degree concept we call *back degree* (Gebremedhin et al., 2013). In an ordering $\pi = v_1, v_2, \ldots, v_n$ of the vertices of a graph $G = (V, E)$, the *back degree* of the vertex v_i is the number of distance-1 neighbors of v_i in G that are ordered *before* v_i in π.

An SL ordering π is defined from highest to lowest, v_n to v_1. Initially, the back degree $b(v)$ of every vertex is equal to its degree $d(v, G)$ in G. The last vertex v_n is a vertex with the *smallest* back degree. With v_n determined, the back degree of every distance-1 neighbor of v_n, by definition, is then the original value minus one. The next vertex in the ordering, v_{n-1}, is a vertex with the smallest back degree among the remaining $n-1$ vertices. Suppose the last $n-i-1$ entries of the ordered vertex set have been determined. The ith vertex in the ordering is then a vertex with the *smallest* back degree among the vertices $U = V \setminus \{v_n, v_{n-1}, \ldots, v_{i+1}\}$ that are yet to be ordered.

An ID ordering π is defined from lowest to highest, v_1 to v_n. Initially, the back degree of every vertex is equal to zero. The first vertex v_1 is a vertex with the *largest* back degree (note that since all back degrees are zero, any one of the vertices would qualify). With v_1 determined, the back degree of every distance-1 neighbor of v_1, by definition, is then the original value plus one. The next vertex in the ordering, v_2, is a vertex with the largest back degree among the remaining $n-1$ vertices. Suppose the first $i-1$ entries of the ordered vertex set have been determined. The ith vertex in the ordering is then a vertex with the *largest* back degree among the vertices $U = V \setminus \{v_1, v_2, \ldots, v_{i-1}\}$ that are yet to be ordered.

Algorithms

We give in Algorithm 4 a template for an *efficient* implementation of the ordering techniques SL and ID. Table 5 shows how the template is specialized to give SL or ID. The sparse, two-dimensional array B in Algorithm 4 is a vehicle used for arriving at efficient implementation. The array, itself implemented as a vector of vectors with total size $|V|$, is used 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 back degree $b(u)$ equal to j. The output of Algorithm 4 is given by the ordered list W of the vertices where $W[i]$ stores the ith vertex in the ordering.

ALGORITHM 4: Template for SL and ID Ordering. B is a sparse, two-dimensional array maintaining *unordered* vertices binned according to their back degrees.

```
   Input: Graph G = (V, E)
   Output: An ordered list W of the vertices in V
1  for each vertex v ∈ V do
2  |  init b(v);
3  |  B[b(v)] ← B[b(v)] ∪ {v};
4  end
5  init i;
6  while check i do
7  |  identify j*, min (or max) index j with B[j] ≠ ∅;
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 ∈ N_1(v) such that w is in B do
12 |  |  B[b(w)] ← B[b(w)] \ {w};
13 |  |  update b(w);
14 |  |  B[b(w)] ← B[b(w)] ∪ {w};
15 |  end
16 |  update i;
17 end
```

We determine the ith vertex in the ordering in constant time by maintaining a pointer to the last element in $B[j^*]$, where j^* is 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 of each vertex can also be performed in constant time (Gebremedhin et al., 2013). Thus the work involved in the ith step of Algorithm 4 is proportional to $d(v, G)$, and the overall complexity of the algorithm is $O(|E|)$.

Applications

The rationale behind the ordering techniques SL and ID in the context of coloring is to bring vertices that are likely to be highly constrained in the choice of colors early in the ordering and thereby reduce the number of colors used. Both of these orderings are highly effective at doing just that. But the use of these orderings is not limited to coloring. For instance, an SL ordering, in reverse order to that obtained by Algorithm 4, can be used to determine *k-cores* (densely connected subgraphs) in social and biological networks. SL ordering, and consequently core computation, is also directly related to the graph-theoretic notions of degeneracy and arboricity (Matula, 1968; Szekeres and Wilf, 1968; Lick

Table 5: How Algorithm 4 specializes to SL or ID.

	SL	**ID**
`init` $b(v)$	$b(v) \leftarrow d(v, G)$	$b(v) \leftarrow 0$
`init` i	$i \leftarrow \lvert V \rvert$	$i \leftarrow 1$
`check` i	$i \geq 1$	$i \leq \lvert V \rvert$
`identify` j^*	$j^* = \min_j \{B[j] \neq \emptyset\}$	$j^* = \max_j \{B[j] \neq \emptyset\}$
`update` $b(w)$	$b(w) \leftarrow b(w) - 1$	$b(w) \leftarrow b(w) + 1$
`update` i	$i \leftarrow i - 1$	$i \leftarrow i + 1$

and White, 1970; Matula et al., 1972; Matula and Beck, 1983; Gebremedhin et al., 2005). Similarly, an ID ordering obtained by Algorithm 4, when reversed, corresponds to an ordering obtained by the *maximum cardinality search* algorithm (Tarjan and Yannakakis, 1984), which is useful in determining chordality of a graph.

Parallelization

We consider two different approaches for the parallelization of the ordering template depicted in Algorithm 4 (Patwary et al., 2011). The first approach aims at parallelizing the ordering template closely maintaining the serial behavior, while the second approach settles for an approximate solution in favor of increased concurrency, thus falling under the Approximate Update paradigm. Both approaches apply equally to SL an ID ordering. To simplify presentation, however, we discuss only the SL case here. We denote by $t(v)$ the thread with which the vertex v is initially associated.

The First Approach: Regular The first task this approach parallelizes is the population of the global bin array B. To achieve this, a *local* two-dimensional array B_t is associated with each thread T_t, $1 \leq t \leq p$. The p local arrays are first populated in parallel. Then, the contents are gathered into the global array B, where the parallelization is now switched to run over bins. The remainder of the algorithm mimics the serial algorithm (Algorithm 4). 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. The approach *Regular* takes advantage of this opportunity and strives to order such vertices and update their neighborhoods in parallel. This gives rise to a variety of *race* conditions. The approach involves careful handling of these, including the use of frequent *atomic* and *critical* statements. Because of the use of these statements the parallel algorithm behaves much like the serial, resulting in poor scalability (Patwary et al., 2011).

The Second Approach: Relaxed The second approach for parallelizing the SL ordering algorithm abandons the use of the global array B altogether and

ALGORITHM 5: A parallel SL ordering algorithm on p threads (Relaxed).

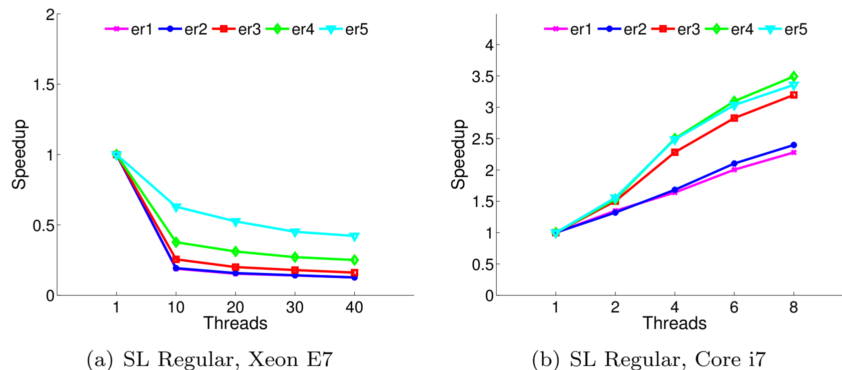
```
   Input: Graph G = (V, E)
   Output: Output: An ordered list W of the vertices in V
 1 for each vertex v ∈ V in parallel do
 2     b(v) ← d(v, G);
 3     B_t(v) [b (v)] ← B_t(v) [b (v)] ∪ {v};
 4 end
 5 i ← |V|;
 6 for t = 1 to p in parallel do
 7     while i ≥ 0 do
 8         Let j* be the smallest index j s.t. B_t [j] ≠ ∅;
 9         Let v be a vertex drawn from B_t [j*];
10         B_t [j*] ← B_t [j*] \ {v};
11         for each vertex w ∈ N_1 (v) do
12             if w ∈ B_t then
13                 B_t [b (w)] ← B_t [b (w)] \ {w};
14                 b (w) ← b (w) − 1;
15                 B_t [b (w)] ← B_t [b (w)] ∪ {w};
16             end
17         end
18         W [i] ← v   (critical);
19         i ← i − 1   (critical);
20     end
21 end
```

works only with the local arrays B_t associated with each thread T_t. In updating locations of neighbors of a vertex, a thread T_t checks whether or not the vertex w desired to be relocated is in the thread's local array B_t. If w is indeed in B_t, 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 *global* ordering. The approach is formalized in Algorithm 5.

Performance Results

The first parallelization approach, Regular, did not scale for a vast majority of the problem-platform combinations in our experiments. As an illustration, we give in Figure 5 speedup plots for the SL-Regular (SL-Reg) algorithm on the two platforms and the `er`-graphs in the dataset; the results on the other two graph classes (`g` and `b`) are similar or worse. In sharp contrast, we found that the approximate update approach Relaxed yielded moderate to excellent speedups as more threads are employed. Figure 6 shows speedup plots for SL-Relaxed (SL-Rel) on the two platforms and all three graph classes.

The plots in figures 5 and 6 show speedups wherein runtimes are *normalized*



(a) SL Regular, Xeon E7

(b) SL Regular, Core i7

Figure 5: Speedup results of the parallel ordering algorithm **SL-Regular** on the two test platforms and for the `er` class of graphs.

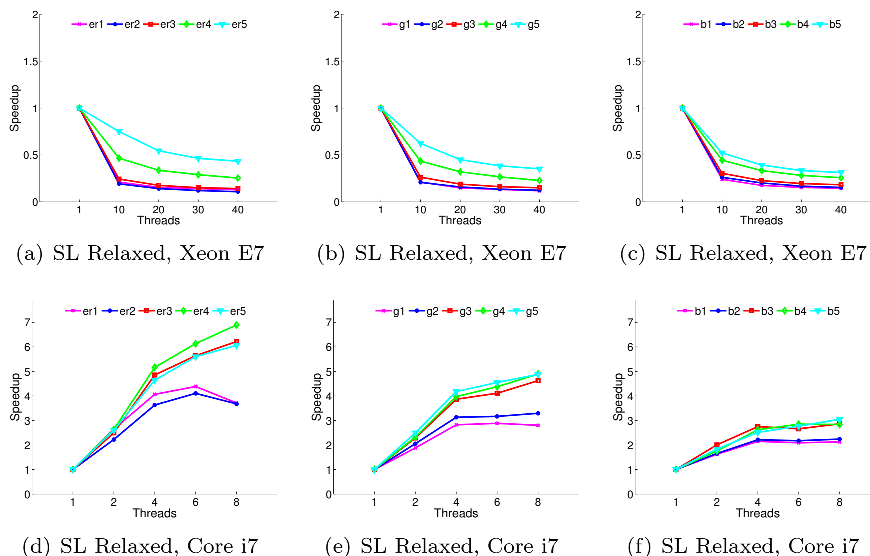
by the parallel algorithm's runtime when one thread is used. Recall that we had provided in Table 4 a summary of the raw compute times in seconds for runs on the Xeon E7 machine of the parallel algorithms on a *single* thread and of the pure sequential algorithms for SL ordering, distance-1 coloring, and distance-2 coloring.

Incidence Degree (ID) ordering: We observed that the *Regular* and *Relaxed* parallelization of ID ordering performed in nearly identical manners as the corresponding parallelizations of SL ordering across both platforms. We therefore omit presenting results on ID ordering.

Summary and Outlook

We introduced two paradigms, called Speculation and Iteration and Approximate Update, that are effective for the parallelization of two frequently used classes of graph algorithms on multi-core architectures. The two classes of graph algorithms are *greedy algorithms* and *vertex ordering procedures*, respectively. We demonstrated the efficacy of the paradigms on two representative algorithms from the class of greedy algorithms (distance-1 and distance-2 coloring) and two representative algorithms from the class of vertex ordering procedures (Smallest Last ordering and Incidence Degree ordering). As test platforms we used two Intel multi-core systems (Xeon E7 and Core i7).

We observe that the Speculation And Iteration paradigm has interesting connections with the theoretical model called *Local Computation Algorithms* (Rubinfeld et al., 2011). We also note that the Approximate Update paradigm can be viewed within the broader theme of design of *concurrent data structures* (Shavit, 2011). Both of these connections are worthwhile directions for future research.



(a) SL Relaxed, Xeon E7 (b) SL Relaxed, Xeon E7 (c) SL Relaxed, Xeon E7

(d) SL Relaxed, Core i7 (e) SL Relaxed, Core i7 (f) SL Relaxed, Core i7

Figure 6: Speedup results of the parallel **SL-Relaxed** ordering algorithm on the two test platforms and for the three classes of graphs `er`, `g` and `b`.

References

Alon, N., Matias, Y., and Szegedy, M. (1999). The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147.

Barker, K. J., Davis, K., Hoisie, A., Kerbyson, D. J., Lang, M., Pakin, S., and Sancho, J. C. (2008). A performance evaluation of the Nehalem quad-core processor for scientific computing. *Parallel Processing Letters*, 18(4):453–469.

Bozdağ, D., Catalyurek, U. V., Gebremedhin, A. H., Manne, F., Boman, E. G., and Ozguner, F. (2010). Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation. *SIAM Journal on Scientific Computing*, 32(4):2418–2446.

Bozdağ, D., Gebremedhin, A. H., Manne, F., Boman, E. G., and Catalyurek, U. V. (2008). A framework for scalable greedy coloring on distributed-memory parallel computers. *Journal of Parallel and Distributed Computing*, 68(4):515–535.

Catalyurek, U., Feo, J., Gebremedhin, A. H., Halappanavar, M., and Pothen, A. (2012). Graph coloring algorithms for multicore and massively multhithreaded architectures. *Parallel Computing*, 38:576–594.

Chakrabarti, D. and Faloutsos, C. (2006). Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1):2.

Coleman, T. F. and More, J. J. (1983). Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 1(20):187–209.

Gebremedhin, A. H. and Manne, F. (2000). Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience*, 12:1131–1146.

Gebremedhin, A. H., Manne, F., and Pothen, A. (2002). Parallel distance-k coloring algorithms for numerical optimization. In *proceedings of Euro-Par 2002*, volume 2400, pages 912–921. Lecture Notes in Computer Science, Springer.

Gebremedhin, A. H., Manne, F., and Pothen, A. (2005). What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705.

Gebremedhin, A. H., Nguyen, D., Patwary, M. M. A., and Pothen, A. (2013). ColPack: Software for graph coloring and related problems in scientific computing. *ACM Transaction on Mathematical Software*. In press.

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing*. Pearson.

Jájá, J. (1992). *An Introduction to Parallel Algorithms*. Addison-Wesley.

Kurzak, J., Bader, D., and Dongara, J. (2010). *Scientific Computing with Multicore and Accelerators*. Chapman and Hall/CRC Press.

Lick, D. R. and White, A. T. (1970). k-degenerate graphs. *Canadian Journal of Mathematics*, 22:1082–1096.

Lin, Y.-L. and Skiena, S. (1995). Algorithms for square roots of graphs. *SIAM J. Discr. Math.*, 8:99–118.

Matula, D. W. (1968). A max-min theorem for graphs with application to graph coloring. *SIAM Rev.*, 10:481–482.

Matula, D. W. and Beck, L. L. (1983). Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM*, 30(3):417–427.

Matula, D. W., Marble, G., and Isaacson, J. (1972). Graph coloring algorithms. In Read, R., editor, *Graph Theory and Computing*, pages 109–122, New York. Academic Press.

McCormick, S. T. (1983). Optimal approximation of sparse hessians and its equivalence to a graph coloring problem. *Math. Programming*, 26:153 − 171.

Pattabiraman, B., Patwary, M. M. A., Gebremedhin, A. H., keng Liao, W., and Choudhary, A. (2013). Fast algorithms for the maximum clique problem on massive sparse graphs. In *WAW13, 10th Workshop on Algorithms and Models for the Web Graph*. To appear.

Patwary, M. M. A., Gebremedhin, A. H., and Pothen, A. (2011). New multithreaded ordering and coloring algorithms for multicore architectures. In Jeannot, E., Namyst, R., and Roman, J., editors, *Proceedings of EuroPar 2011*, volume 6853 of *Lecture Notes in Computer Science*, pages 250–262. Springer.

Patwary, M. M. A., Refsnes, P., and Manne, F. (2012). Multi-core spanning forest algorithms using disjoint-set data structure. In *IPDPS 2012*, pages 827–835.

Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., and Sui, X. (2011). The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 12–25, New York, NY, USA. ACM.

Rubinfeld, R., Tamir, G., Vardi, S., and Xie, N. (2011). Fast local computation algorithms. In *International Conference on Supercomputing (ICS 2011*, pages 496–508.

Sariyuce, A. E., Saule, E., and Catalyurek, U. V. (2011). Improving graph coloring on distributed memory parallel computers. In *Proc. of HiPC 2011*.

Sariyuce, A. E., Saule, E., and Catalyurek, U. V. (2012). Scalable hybrid implementation of graph coloring using MPI and OpenMP. In *Proc. IPDPS Workshops and PhD Forum, Workshop on Parallel Computing and Optimization (PCO'12)*.

Shavit, N. (2011). Data structures in the multicore age. *Communications of the ACM*, 54:76–84. DOI: 10.1145/1897852.1897873.

Szekeres, G. and Wilf, H. S. (1968). An inequality for the chromatic number of a graph. *Journal of Combinatorial Theory*, 4:1–3.

Tarjan, R. E. and Yannakakis, M. (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Scientific Computing*, 13(3):566–579.

Tian, C., Feng, M., Nagarajan, V., and Gupta, R. (2009). Speculative parallelization of sequential loops on multicores. *Int J Parallel Prog*, 37(1):508–535.

Zavlanos, M. M., Spesivtsev, L., and Pappas, G. J. (2008). A distributed auction algorithm for the assignment problem. In *Proceedings of the 47th IEEE Conference on Decision and Control*, pages 1212–1217.