Parallel Computing 38 (2012) 576–594

Contents lists available at SciVerse ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Graph coloring algorithms for multi-core and massively multithreaded architectures

Ümit V. Çatalyürek [a], John Feo [b], Assefaw H. Gebremedhin [c,*], Mahantesh Halappanavar [b], Alex Pothen [c]

[a] Departments of Biomedical Informatics and Electrical & Computer Engineering, The Ohio State University, United States
[b] Pacific Northwest National Laboratory, United States
[c] Department of Computer Science, Purdue University, United States

ARTICLE INFO

Article history:
Received 19 February 2011
Received in revised form 15 May 2012
Accepted 15 July 2012
Available online 31 July 2012

Keywords:
Multi-core/multithreaded computing
Parallel graph algorithms
Combinatorial scientific computing
Graph coloring

ABSTRACT

We explore the interplay between *architectures* and *algorithm design* in the context of shared-memory platforms and a specific graph problem of central importance in scientific and high-performance computing, distance-1 graph coloring. We introduce two different kinds of multithreaded heuristic algorithms for the stated, NP-hard, problem. The first algorithm relies on *speculation* and *iteration*, and is suitable for *any* shared-memory system. The second algorithm uses *dataflow* principles, and is targeted at the non-conventional, massively multithreaded Cray XMT system. We study the performance of the algorithms on the Cray XMT and two multi-core systems, Sun Niagara 2 and Intel Nehalem. Together, the three systems represent a spectrum of multithreading capabilities and memory structure. As testbed, we use synthetically generated large-scale graphs carefully chosen to cover a wide range of input types. The results show that the algorithms have scalable runtime performance and use nearly the same number of colors as the underlying serial algorithm, which in turn is effective in practice. The study provides insight into the design of high performance algorithms for irregular problems on many-core architectures.

1. Introduction

Graph problems frequently arise in many practical applications, including computational science and engineering, data mining, and data analysis. When the applications are large-scale, solutions need to be obtained on parallel computing platforms. High performance and good scalability are hard-to-achieve on graph algorithms, for a number of well-recognized reasons [1]: runtime is dominated by memory latency rather than processor speed, and typically there is little (or no) computation involved to hide memory access costs. Access patterns are irregular and are determined by the structure of the input graph, rendering prefetching techniques inapplicable. Data locality is poor, making it difficult to obtain good memory system performance. While concurrency can be abundant, it is often fine-grained, requiring concurrent processes to synchronize at individual vertices or edges.

For these reasons, graph algorithms that perform and scale well on distributed memory machines are relatively small in number and kind. More success stories have been reported on shared memory platforms, and interest in these platforms is growing with the increasing preponderance, popularity, and sophistication of *multi-core* architectures [2–4].

* Corresponding author. Address: Department of Computer Science, Purdue University, West Lafayette, Indiana 47907, United States. Tel.: +1 765 496 1761; fax: +1 765 494 0739.
E-mail address: agebreme@purdue.edu (A.H. Gebremedhin).

0167-8191/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2012.07.001

The primary mechanism for tolerating memory latencies on most shared memory systems is the use of *caches*, but caches have been found rather ineffective for many graph algorithms. A more effective mechanism is *multithreading*. By maintaining multiple threads per core and switching among them in the event of a long latency operation, a multithreaded processor uses parallelism to hide latencies. Whereas caches "hide" only memory latencies, thread parallelism can hide both memory and synchronization overheads. Thus, multithreaded, shared memory systems are more suitable platforms for many graph algorithms than either distributed memory machines or single-threaded, multi-core shared memory systems.

We explore in this paper the interplay between architectures and algorithm design in the context of shared-memory multithreaded systems and a specific graph problem, *distance-1 graph coloring*.

Graph coloring in a generic sense is an abstraction for partitioning a set of binary-related objects into subsets of independent objects. A need for such a partitioning arises in many situations where there is a scarce resource that needs to be utilized optimally. One example of a broad area of application is in discovering concurrency in parallel computing, where coloring is used to identify subtasks that can be carried out or data elements that can be updated simultaneously [5–7]. On emerging heterogenous architectures, coloring algorithms on certain "interface-graphs" are used at runtime to decompose computation into concurrent tasks that can be mapped to different processing units [8]. Another example of a broad application area of coloring is the efficient computation of sparse derivative matrices [9–11].

Distance-1 coloring is known to be NP-hard not only to solve optimally but also in an approximate sense [12]. However, *greedy* algorithms, which run in linear time in the size of the graph, often yield near optimal solutions on graphs that arise in practice, especially when the greedy algorithm is initialized with careful *vertex ordering* techniques [9,13]. In contexts where coloring is used as a step to enable some overarching computation, rather than being an end in itself, greedy coloring algorithms are attractive alternatives to slower, local-improvement type heuristics because they yield sufficiently small number of colors while using run times that are much shorter than the computations they enable.

Contributions. This paper is concerned with the effective parallelization of greedy distance-1 coloring algorithms on multithreaded architectures. To that end, we introduce two different kinds of multithreaded algorithms targeted at two different classes of architectures.

The first multithreaded algorithm is suitable for *any* shared memory system, including the emerging and rapidly evolving multi-core platforms. The algorithm relies on *speculation* and *iteration*, and is derived from the parallelization framework for graph coloring on distributed memory architectures developed in Bozdağ et al. [14,15]. We study the performance of the algorithm on three different platforms, an Intel Nehalem, a Sun Niagara 2, and a Cray XMT. These three systems employ multithreading—in varying degrees—to hide latencies; the former two additionally rely on cache hierarchies to hide latency. The amount of concurrency explored in the study ranges from small (16 threads on the Nehalem) to medium (128 threads on the Niagara) to massive (16,384 threads on the XMT). We find that the limited parallelism and coarse synchronization of the iterative algorithm fit well with the limited mulithreading capabilities of the Intel and Sun systems.

The iterative algorithm runs equally well on the XMT, but it does not take advantage of the system's massively multithreaded processors and hardware support for fast synchronization. To better exploit the XMT's unique hardware features, we developed a fine-grained, *dataflow* algorithm making use of the single-word synchronization mechanisms available on the XMT. The dataflow algorithm is the second algorithm presented in this paper.

The performance study is conducted using a set of carefully chosen synthetic graphs representing a wide spectrum of input types. The results show that the algorithms perform and scale well on massive graphs containing as many as a billion edges. This is true of the dataflow algorithm on the Cray XMT and of the iterative algorithm on all three of the platforms considered; on the XMT, the dataflow algorithm runs faster and scales better than the iterative algorithm. The number of colors the algorithms use is nearly the same as that used by the underlying serial algorithm.

In the remainder of this section, we briefly discuss related work. The rest of the paper is organized as follows. To establish background, especially for the design of the dataflow algorithm, we review in Section 2 basic architectural features of the three platforms used in the study. We describe the coloring algorithms in detail and discuss their relationship with previous work in Section 3. In Section 4, we discuss the rationale for, the generation of, and the characteristics of the synthetic graphs used in the study. The experimental results are presented and analyzed in detail in Section 5. We end by drawing broader conclusions in Section 6.

Related work. Graph coloring has a vast literature, and various approaches have been taken to solve coloring problems on computers. Exact algorithms include those based on integer programming, semidefinite programming, and constraint programming. Heuristic approaches include the greedy algorithms mentioned earlier, local search algorithms, population-based methods, and methods based on successively finding independent sets in the graph [16]. There have been two DIMACS implementation challenges on coloring [17], where the focus was on obtaining the smallest number of colors possible for any problem instance; algorithms merely focused on this objective typically have large running times, and as a result they can handle only small instances of graphs, often with at most a thousand vertices.

A different approach has been taken by Turner [18], who analyzes a heuristic algorithm due to Brélaz [19]. Turner considers random graphs that have the property that they can be colored with k colors, where k is a constant, and shows that Brélaz's algorithm colors such graphs with high probability using k colors. Further analysis of this phenomenon is provided by Coja-Oghlan et al. [20]. An experimental study we conducted in a related work [13] has shown that on many graphs from various application areas, a greedy coloring algorithm, initialized with appropriate vertex ordering techniques, yields fewer colors than Brélaz's algorithm, while running faster. A survey of several graph coloring problems (e.g., distance-1, distance-2, star, and acyclic coloring on general graphs; and partial distance-2 coloring on bipartite graphs) as they arise in the context

of automatic differentiation is provided in Gebremedhin et al. [11]. Effective heuristic algorithms for the star and acyclic coloring problems and case studies demonstrating the use of those coloring algorithms in sparse Hessian computation have been presented in subsequent works [21,22].

2. Architectures

The three platforms considered in this study—an Intel Nehalem, a Sun Niagara 2, and a Cray XMT—represent a broad spectrum of multithreading capabilities and memory structure. The Intel Nehalem relies primarily on a cache-based hierarchical memory system as a means for hiding latencies, supporting just two threads per core. The Cray XMT has a flat, cache-less memory system and uses massive multithreading as the sole mechanism for tolerating latencies in data access. The Sun Niagara 2 offers a middle path by using a moderate number of hardware-threads along with a hierarchical memory system. We review in the remainder of this section some of the specific architectural features and programming abstractions available in the three systems, focusing on aspects relevant to our work on algorithm design and performance evaluation.

2.1. The Intel Nehalem

The Intel system we considered, Nehalem EP (Xeon E5540), consists of two quad-core Gainestown processors running at 2.53 GHz base core frequency. See Fig. 1 for a block diagram. Each core supports two threads and has one instruction pipeline. The multithreading variant on this platform is *simultaneous*, which means multiple instructions from ready threads are executed in a given cycle.

The system has 24 GB total memory. The memory system per core consists of: 32 kB, 2-way associative I1 (Instruction) cache; 32 kB, 8-way associative L1 cache; 2×6 MB, 8-way associative L2 cache; and 8 MB L3 cache shared by the four cores. The maximum memory bandwidth of the system is 25.6 GB/s, the on-chip latency is 65 cycles, and the latency between processors is 106 cycles.

On Nehalem, the quest for good performance is addressed not only via caching, that is, exploitation of spatial and temporal locality, but also through various advanced architectural features. The advanced features include: loop stream detection (recognizing instructions belonging to loops and avoiding branch prediction in this context); a new MESIF cache coherency protocol that reduces coherency control traffic; two branch-target buffers for improved branch prediction; and out-of-order execution. In MESIF, the states are Modified, Exclusive, Shared, Invalid, and a new Forward state; only one instance of a cache line can be in the Forward state, and it is responsible for responding to read requests for the cache line, while the cache lines in the Shared state remain silent, reducing coherency traffic. For further architectural details on the Nehalem, see, for example, the paper [23].

2.2. The Sun Niagara 2

The Sun UltraSPARC T2 platform, Niagara 2, has two 8-core sockets with each core supporting eight hardware threads. Fig. 2 shows a block diagram. Similar to the Nehalem, multithreading on the Niagara 2 is *simultaneous*. The system has a

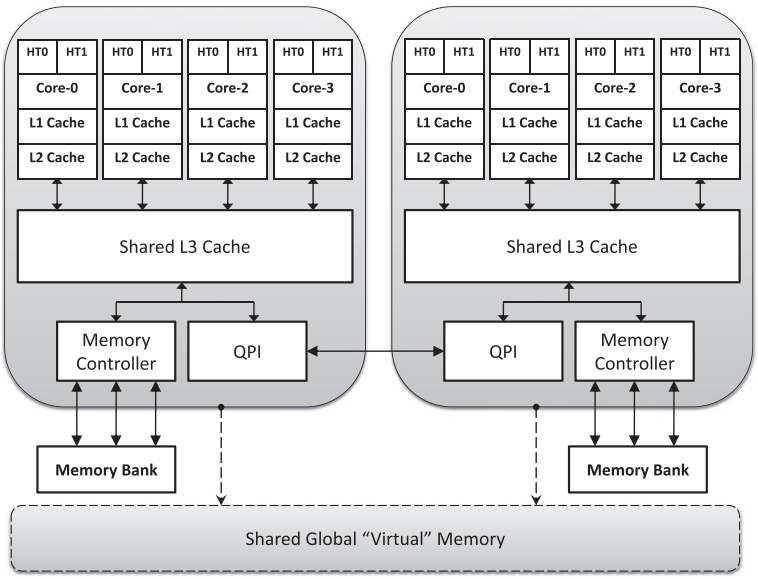


Fig. 1. Block diagram of the Intel Nehalem.

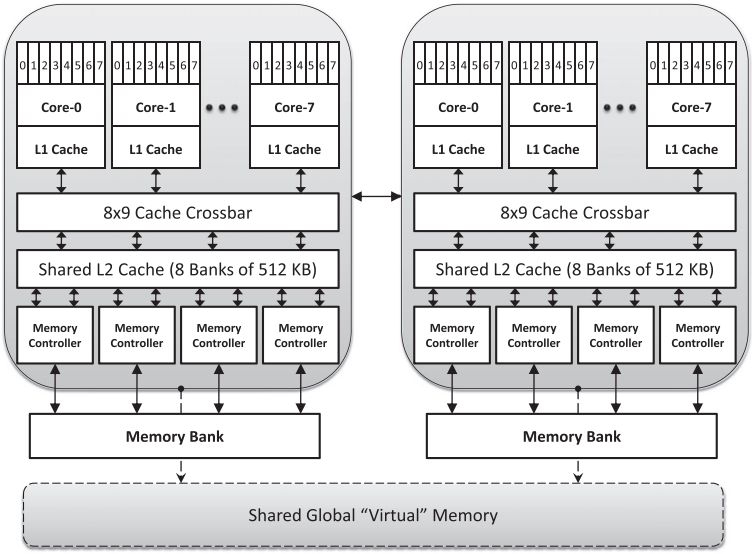


Fig. 2. Block diagram of the Sun UltraSparc T2 (Niagara 2).

shallow instruction pipeline (in contrast to the XMT), and each core has two integer execution units, a load/store unit, and a floating-point unit. The pipeline is capable of issuing two instructions per cycle, either from the same thread or from different threads. Threads are divided into two groups of four, and one instruction from each group may be selected based on the *least-recently fetched* policy on ready threads. The clock frequency of the processor is 1.165 GHz.

The total size of the memory system is 32 GB. Each core has an 8 kB, 4-way associative L1 cache for data and a 16 kB, 8-way associative I1 cache for instructions. Unlike Nehalem, where L2 cache is private, Niagara 2 has a shared, 16-way associative L2 cache of size 4 MB. The cache is arranged in 8 banks and is shared using a crossbar switch between CPUs. The latency is 129 cycles for local memory accesses and 193 cycles for remote memory accesses. The peak memory bandwidth is 50 GB/s for reads and 26 GB/s for writes. See [24] for additional details.

2.3. The Cray XMT

The Cray XMT platform used in this study is comprised of 128 Cray Threadstorm (MTA-2) processors interconnected via a 3D torus. Fig. 3 shows a block diagram of the platform from a programmer's point of view. Each processor has 128 hardware thread-streams and one instruction pipeline. Each thread-stream is equipped with 32 general purpose registers, 8 target registers, a status word, and a program counter. Consequently, each processor can maintain up to 128 separate software threads. In every cycle, a processor context switches among threads with ready instructions in a fair manner choosing one of the threads to issue its next instruction. In other words, the multithreading variant on the XMT, in contrast to Nehalem and Niagara 2, is *interleaved*. A processor stalls only if no thread has a ready instruction. There are three functional units, M, A, and C for executing the instructions. On every cycle, the M-unit can issue a read or write operation, the A-unit can execute a fused multiply-add, and the C-unit can execute either a control or an add operation. Instruction execution on the XMT is deeply pipelined, with a 21-stage pipeline. The clock frequency is 500 MHz.

2.3.1. Memory system

The memory system of the XMT is cache-less, and is globally accessible by all processors. The system supports a shared memory programming model. The global address space is built from physically distributed memory modules of 8 GB per processor, yielding a total memory size of 1 TB. Further, the address space is built using a hardware *hashing* mechanism that maps the data randomly to the memory modules, in blocks of size 64 bytes, to minimize conflicts and hot-spots. All memory words are 8 bytes wide and the memory is byte-addressable. The worst-case memory latency is approximately 1000 cycles, and the average value is around 600 cycles. The sustainable minimum remote memory reference rate is 0.52 GB/s per processor for random reads and 0.26 GB/s per processor for random writes. Memory access rates scale with the number of processors.

2.3.2. Fine-grained synchronization

Associated with every word are several additional bits—a full/empty bit, a pointer forwarding bit, and two trap bits. These are used for fine-grained synchronization. The XMT provides many efficient extended memory semantic operations for

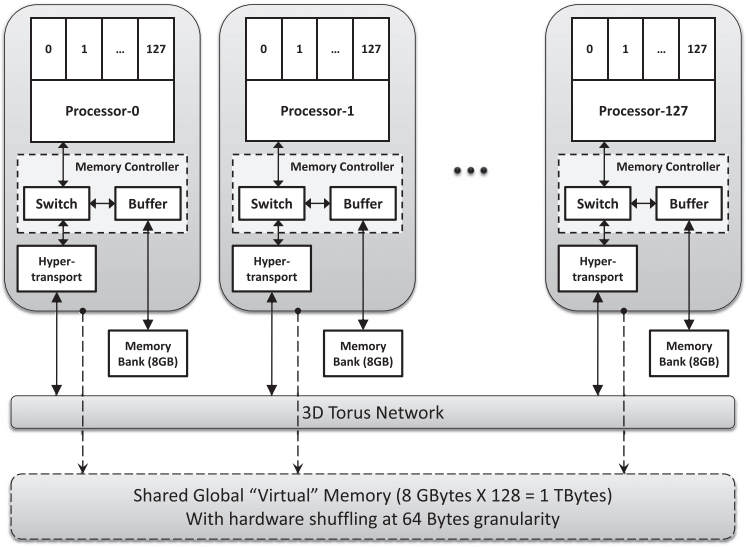


Fig. 3. Block diagram of the Cray XMT.

Table 1
Overview of architectural features of the platforms Nehalem, Niagara 2 and XMT.

	Nehalem	Niagara 2	Cray XMT
Clock (GHz)	2.53	1.165	0.5
Sockets	2	2	128
Cores/socket	4	8	–
Threads/core	2	8	128
Threads total	16	128	16,384
Multithreading	Simultaneous	Simultaneous	Interleaved
Memory (GB)	24	32	1024
Cache	L1/L2, shared L3	L1, shared L2	Cache-less, flat
Bandwidth	Max	Peak	Remote
	25.6 GB/s	50 GB/s (read),	0.52 GB/s/proc (read),
		26 GB/s (write)	0.26 GB/s/proc (write)
Latency	65 cycles (on-chip),	129 cycles (local),	600 cycles (average),
	106 cycles (b/n proc)	193 cycles (remote)	1000 cycles (worst-case)

manipulating the full/empty bits in one clock cycle. Listed below are examples of functions effecting such semantic operations—many of these functions are used in the XMT-specific coloring algorithms we have developed:

`purge` sets the full/empty bit to empty and assigns a zero value to the memory location of the associated word.
`readff` reads a memory location only when the full/empty bit is full and leaves the bit full when the read completes.
`readfe` reads a memory location only when the full/empty bit is full and leaves the bit empty when the read completes.
`writeef` writes to a variable only if the full/empty bit is empty and leaves the bit full when the write completes.

If any extended memory semantic operation finds the full/empty bit in the wrong state, the operation waits. The request returns to the issuing processor and is placed in a queue to be retried. Note that the instruction is issued only once, and while the instruction is being retried, the processor continues to issue instructions from other threads. See [25,26] for further details about the XMT platform.

2.4. Quick comparison

Table 1 provides a quick comparative summary of some of the key architectural features of the three platforms discussed in Sections 2.1–2.3. Note that the bandwidth and latency numbers listed in Table 1 are *peak values*. With quick experiments on the Nehalem and Niagara 2 systems using the benchmarking tool *libmicro*, the bandwidth and latency numbers we observed were roughly about a factor of two worse than those listed in Table 1.

3. The algorithms

We describe in this section the multithreaded distance-1 coloring algorithms we developed for architectures such as those discussed in Section 2. We begin the section with a quick review of the problem, the underlying serial algorithm we use to solve it, and prior work on parallelization.

3.1. Background

3.1.1. The problem

Graph coloring problems come in several variations [11]. The variant considered in this paper is *distance-1 coloring*, an assignment of positive integers (called colors) to vertices such that adjacent vertices receive different colors. The objective of the associated problem is to minimize the number of colors used. The problem has long been known to be NP-hard to solve optimally. Recently, it has been shown that, for all $\epsilon > 0$, the problem remains NP-hard to *approximate* to within $n^{1-\epsilon}$, where n is the number of vertices in the graph [12].

Algorithm 1. A sequential greedy coloring algorithm

```
1: procedure GREEDY(G(V, E))
2:   Initialize data structures
3:   for each v ∈ V do
4:     for each w ∈ adj(v) do
5:       forbiddenColors[color[w]] ← v          ▷ mark color of w as forbidden to v
6:     c ← min{i > 0 : forbiddenColors[i] ≠ v}  ▷ smallest permissible color
7:     color[v] ← c
```

3.1.2. An effective algorithm

Despite these pessimistic theoretical results, for many graphs or classes of graphs that occur in practice, solutions that are provably optimal or near-optimal can be obtained using a *greedy* (aka sequential) algorithm. The greedy algorithm runs through the set of vertices in some *order*, at each step assigning a vertex the *smallest* permissible color. We give a formal presentation of an *efficient* formulation of the greedy algorithm in procedure GREEDY (Algorithm 1). The formal presentation is needed since the procedure forms the foundation for the parallel algorithms presented in this paper.

In Algorithm 1, and in other algorithms specified later in this paper, $adj(v)$ denotes the set of vertices adjacent to the vertex v, color is a vertex-indexed array that stores the color of each vertex, and forbiddenColors is a color-indexed array used to mark the colors that are impermissible to a particular vertex. The array color is initialized at the beginning of the procedure with each entry color[w] set to zero, to indicate that vertex w is not yet colored. Each entry of the array forbiddenColors is initialized *once* at the beginning of the procedure with some value $a \notin V$. By the end of the inner for-loop in Algorithm 1, all of the colors that are impermissible to the vertex v are recorded in the array forbiddenColors. In line 6, the array forbiddenColors is scanned from left to right in search of the lowest positive index i at which a value different from v, the vertex being colored, is encountered; this index corresponds to the smallest permissible color c to the vertex v. The color assignment is done in line 7.

Note that since the colors impermissible to the vertex v are marked in the array forbiddenColors using v as a label, rather than say a boolean flag, the array forbiddenColors does not need to be re-initialized in every iteration of the loop over the vertex set V. Further, the search for color in line 6 terminates after at most $d(v)+1$ attempts, where $d(v) = |adj(v)|$ is the degree of vertex v. Therefore, the work done while coloring a vertex v is proportional to its degree, independent of the size of the array forbiddenColors. Thus the time complexity of GREEDY is $O(|V| + |E|)$, or simply $O(|E|)$ if the graph is connected. It can also be seen that the number of colors used by the algorithm is never larger—and often significantly smaller—than $\Delta + 1$, where Δ is the maximum degree in the graph. With good vertex *ordering* techniques, the number of colors used by GREEDY is in fact often near-optimal for practical graph structures and always bounded by $B + 1$, where $B \leqslant \Delta$ is the maximum *back degree*—number of already colored neighbors of a vertex—in the vertex ordering used by GREEDY [13]. A consequence of the fact that the number of colors used by GREEDY is bounded by $\Delta + 1$ is that the array forbiddenColors need not be of size larger than that bound.

3.1.3. Parallelizing the greedy algorithm

Because of its sequential nature, GREEDY is challenging to parallelize, when one also wants to keep the number of colors used close to the serial case. A number of approaches have been investigated in the past to tackle this issue. One class of the investigated approaches relies on iteratively finding a *maximal independent set* of vertices in a progressively shrinking graph and coloring the vertices in the independent set in parallel. In many of the methods in this class, the independent set itself is computed in parallel using some variant of Luby's algorithm [27]. An example of a method developed along this direction is the work of Jones and Plassmann [28]. Finocchi et al. [29] also follow this direction, but enhance their algorithm in many other ways as well.

Gebremedhin and Manne [30] proposed *speculation* as an alternative strategy for coping with the inherent sequentiality of the greedy algorithm. The idea here is to abandon the requirement that vertices that are colored concurrently form an

independent set, and instead color as many vertices as possible concurrently, tentatively tolerating potential conflicts, and detect and resolve conflicts afterwards. A basic shared-memory algorithm based on this strategy is given in [30]. The main steps of the algorithm are to distribute the vertex set equally among the available processors, let each processor speculatively color its vertices using information about already colored vertices (Phase 1), detect eventual conflicts in parallel (Phase 2), and finally re-color vertices involved in conflicts sequentially (Phase 3).

Bozdağ et al. [14] extended the algorithm in [30] in a variety of ways to make it suitable for and well-performing on distributed memory architectures. One of the extensions is replacing the sequential re-coloring phase with a parallel iterative procedure. Many of the other extensions are driven by performance needs in a distributed-memory setting: the graph needs to be *partitioned* among processors in a manner that minimizes communication cost and maximizes concurrency; the speculative coloring phase is better when organized in a *bulk synchronous* fashion where computation and communication are coarse-grained; etc. In addition, they investigate a variety of techniques for choosing colors for vertices and relative ordering of *interior* and *boundary* vertices, a distinction that arises due to the partitioning among processors. When all the ingredients are put together, it was shown that the approach based on speculation and iteration outperforms independent set-based approaches. The framework of [14] is used for the design of a distributed-memory parallel algorithm for distance-2 coloring in [15].

3.2. Iterative parallel coloring algorithm

In this paper, we adapt the distributed-memory iterative parallel coloring algorithm developed in [14] to the context of multithreaded, shared-memory platforms. In the distributed-memory setting, the parallel coloring algorithm relies on graph partitioning to distribute and statically map the vertices and coloring tasks to the processors, whereas in the multithreaded case the vertex coloring tasks are scheduled on processors from a pool of threads created with appropriate programming constructs.

Algorithm 2. An iterative parallel greedy coloring algorithm

```
1:  procedure ITERATIVE(G(V, E))
2:    Initialize data structures
3:    U ← V                                          ▷ U is the current set of vertices to be colored
4:    while U ≠ ∅ do
5:      for each v ∈ U in parallel do                ▷ Phase 1: tentative coloring
6:        for each w ∈ adj(v) do
7:          forbiddenColors[color[w]] ← v            ▷ thread-private
8:        c ← min{i > 0 : forbiddenColors[i] ≠ v}
9:        color[v] ← c
10:     R ← ∅                                        ▷ R is a set of vertices to be re-colored
11:     for each v ∈ U in parallel do                ▷ Phase 2: conflict detection
12:       for each w ∈ adj(v) do
13:       if color[v] = color[w] and v > w then
14:         R ← R ∪ {v}
15:     U ← R
```

The algorithm we use here, formally outlined in Algorithm 2, proceeds in rounds. Each round has two phases, a *tentative coloring* phase and a *conflict detection* phase, and each phase is executed in parallel over a relevant set of vertices (see lines 5 and 11). In our implementation, the loops in lines 5 and 11 are parallelized using the OpenMP directive `#pragma omp parallel for`, and those parallel loops can be scheduled in any manner. The tentative coloring (first) phase is essentially the same as Algorithm 1, except that it is concurrently run by multiple threads; in Algorithm 2, the array forbiddenColors is *private* to each thread. In the conflict-detection (second) phase, each thread examines a relevant subset of vertices that are colored in the current round for consistency and identifies a set of vertices that needs to be re-colored in the next round to resolve any detected conflicts— a conflict is said to have occurred when two adjacent vertices get the same color. Given a pair of adjacent vertices involved in a conflict, it suffices to recolor only one of the two to resolve the conflict. In Algorithm 2, as can be seen in line 13, the vertex with the higher *index* value is chosen to be re-colored in the event of a conflict. Algorithm 2 terminates when no more vertices to be re-colored are left.

Assuming that the number of rounds required by Algorithm 2 is sufficiently small, the overall work performed by the algorithm is linear in the input size. This assumption is realistic and is supported empirically by the experimental results to be presented in Section 5.

3.3. Cray XMT-specific algorithms

We implemented Algorithm 2 on the Cray XMT by replacing the OpenMP directive `#pragma omp parallel for` used to parallelize lines 5 and 11 with the XMT directive `#pragma mta assert parallel`. Typically, parallel algorithms written for conventional processors are too coarse-grained to effectively use the massive multithreading capability provided by the

XMT; hence, we were motivated to explore the potential of an alternative algorithm that employs *fine-grained* synchronization of threads and avoids the occurrence of conflicts (hence the need for re-coloring) in greedy coloring.

3.3.1. A dataflow algorithm

Algorithm 3. A dataflow algorithm for coloring

```
1: procedure DATAFLOW(G(V, E))
2:   for each v ∈ V in parallel do
3:     purge(color[v])                    ▷ Sets full/empty bit to empty and value to zero
4:   for each v ∈ V in parallel do
5:     for each w ∈ adj(v) where w < v do
6:       c_w ← readff(color[w])           ▷ Wait until full/empty bit becomes full
7:       forbiddenColors[c_w] ← v         ▷ thread-private
8:     c ← min{i > 0 : forbiddenColors[i] ≠ v}
9:     writeef(color[v], c)               ▷ Write color and set full/empty bit to full
```

Using the extended memory semantic operations of the XMT functions discussed in Section 2.3 for fine-grained synchronization of threads, we developed a dataflow algorithm for coloring, which is formally outlined in DATAFLOW (Algorithm 3). The algorithm consists of two parallel loops. The first loop *purges* the array used to store the color of each vertex. The second loop runs over all vertices with thread T_v responsible for coloring vertex v. It has two steps. First, the thread T_v reads the color chosen by the neighbors of v with *higher precedence*. We use the *indices* of the vertices to define the precedence order such that vertices with smaller indices will have higher precedence. Second, having read the colors, the thread T_v chooses the smallest permissible color and assigns it to the vertex v. Because a `readff` is used to read the colors of neighbors of v and a `writeef` to write the color of v, the algorithm is race-free. Leaving the XMT-specific mechanism in which concurrent execution is made possible aside, note that this algorithm is conceptually the same as the Jones–Plassmann algorithm [28].

3.3.2. A recursive dataflow algorithm

As written, the algorithm DATAFLOW may deadlock on a system with fewer compute resources than vertices, since only a subset of the threads can start; the others need to wait for resources to become available. If all active threads read vertices whose threads are waiting, deadlock occurs.

The deadlock can be eliminated in one of three ways: (1) preemption, (2) ordering, or (3) recursion. Preempting blocked threads is expensive, and is a poor choice when high-performance and scalability are important goals. Ordering the threads to ensure progress may not be possible in all cases, and in many cases, computing a correct order is expensive. Using recursion is a simple and often efficient way to eliminate deadlock in dataflow algorithms executing on systems with limited compute resources.

We outline the deadlock-free recursive variant of the dataflow coloring algorithm in procedure DATAFLOWRECURSIVE (Algorithm 4). The function PROCESSVERTEX (Algorithm 5) is now responsible for coloring vertex v. Its code is similar to the body of the second loop of DATAFLOW (Algorithm 3). Before reading a neighbor's color, the procedure checks whether or not the neighbor is processed. If the neighbor is processed, the procedure issues a `readff` to read its color; if the neighbor is not processed, the procedure processes the neighbor itself. The recursive algorithm stores the state of each vertex in a new array named state; a value of zero indicates that a vertex is unprocessed. The algorithm also utilizes the XMT intrinsic function `int_fetch_add` to manipulate the state of each vertex. The function is an atomic operation that increments the value of a given memory location with a given constant and returns the *original* value. Note that the procedure PROCESSVERTEX is called only once per vertex—when its state is zero. Hence the overall work performed by the algorithm is linear in the input size.

Algorithm 4. Recursive dataflow algorithm for coloring

```
1: procedure DATAFLOWRECURSIVE(G(V, E))
2:   for each v ∈ V in parallel do
3:     purge(color[v])                    ▷ Sets full/empty bit to empty and value to zero
4:     state[v] ← 0
5:   for each v ∈ V in parallel do
6:     CurState ← int_fetch_add(state[v], 1)   ▷ returns 0 if first time
7:     if CurState = 0 then
8:       PROCESSVERTEX(v)
```

Algorithm 5. A routine called by DataflowRecursive (Algorithm 4)

```
1: procedure ProcessVertex(v)
2:   for each w ∈ adj(v) where w < v do
3:     CurState ← int_fetch_add(state[w], 1)
4:     if CurState = 0 then
5:       ProcessVertex(w)                                  ▷ Recursive call
6:     c_w ← readff(color[w])                              ▷ Wait until full/empty bit becomes full
7:     forbiddenColors[c_w] ← v                            ▷ thread-private
8:   c ← min{i > 0 : forbiddenColors[i] ≠ v}
9:   writeef(color[v], c)                                  ▷ Write color and set full/empty bit to full
```

4. Test graphs

We studied the performance of algorithms Iterative and DataflowRecursive on the three platforms discussed in Section 2 using a set of *synthetically generated* graphs. The test graphs are designed to represent a wide spectrum of input types. We briefly discuss in this section the algorithm used to generate the test graphs and the key structural properties (relevant to the performance of the coloring algorithms) of the generated graphs.

4.1. The graph generator

The test graphs were created using the R-MAT graph generator [31]. Let the graph to be generated consist of $|V|$ vertices and $|E|$ edges. The R-MAT algorithm works by recursively subdividing the adjacency matrix of the graph to be generated (a $|V|$ by $|V|$ matrix) into four quadrants $(1,1),(1,2),(2,1)$, and $(2,2)$, and distributing the $|E|$ edges within the quadrants with specified probabilities. The distribution is determined by four non-negative parameters (a,b,c,d) whose sum equals one. Initially every entry of the adjacency matrix is zero (no edges added). The algorithm places an edge in the matrix by choosing one of the four quadrants $(1,1),(1,2),(2,1)$, or, $(2,2)$ with probabilities a,b,c, or, d, respectively. The chosen quadrant is then subdivided into four smaller partitions and the procedure is repeated until a 1×1 quadrant is reached, where the entry is incremented (that is, the edge is placed). The algorithm repeats the edge generation process $|E|$ times to create the desired graph $G=(V,E)$.

By choosing the four parameters appropriately, graphs of varying characteristics can be generated. We generated three graphs of the same size but widely varying structures by using the following set of parameters: $(0.25,0.25,0.25,0.25)$; $(0.45,0.15,0.15,0.25)$; $(0.55,0.15,0.15,0.15)$.

We call the three graphs RMAT-ER, RMAT-G, and RMAT-B, respectively. (The suffix ER stands for "Erdős-Rényi," G and B stand for "good" and "bad" for reasons that will be apparent shortly.) Table 2 provides basic statistics on the structures of the three graphs. The graphs were generated on the XMT using an implementation of the R-MAT algorithm similar to the one described in [32]. The graphs were then saved on disk for reuse on other platforms. Duplicate edges and self loops were removed; the small variation in the number of edges is due to such removals. The sizes of the graphs were chosen so that they would fit on all three of the platforms we consider. For scalability study on the XMT, however, we generated *larger* graphs using the same three sets of four parameters; we shall encounter these in Section 5, Table 4.

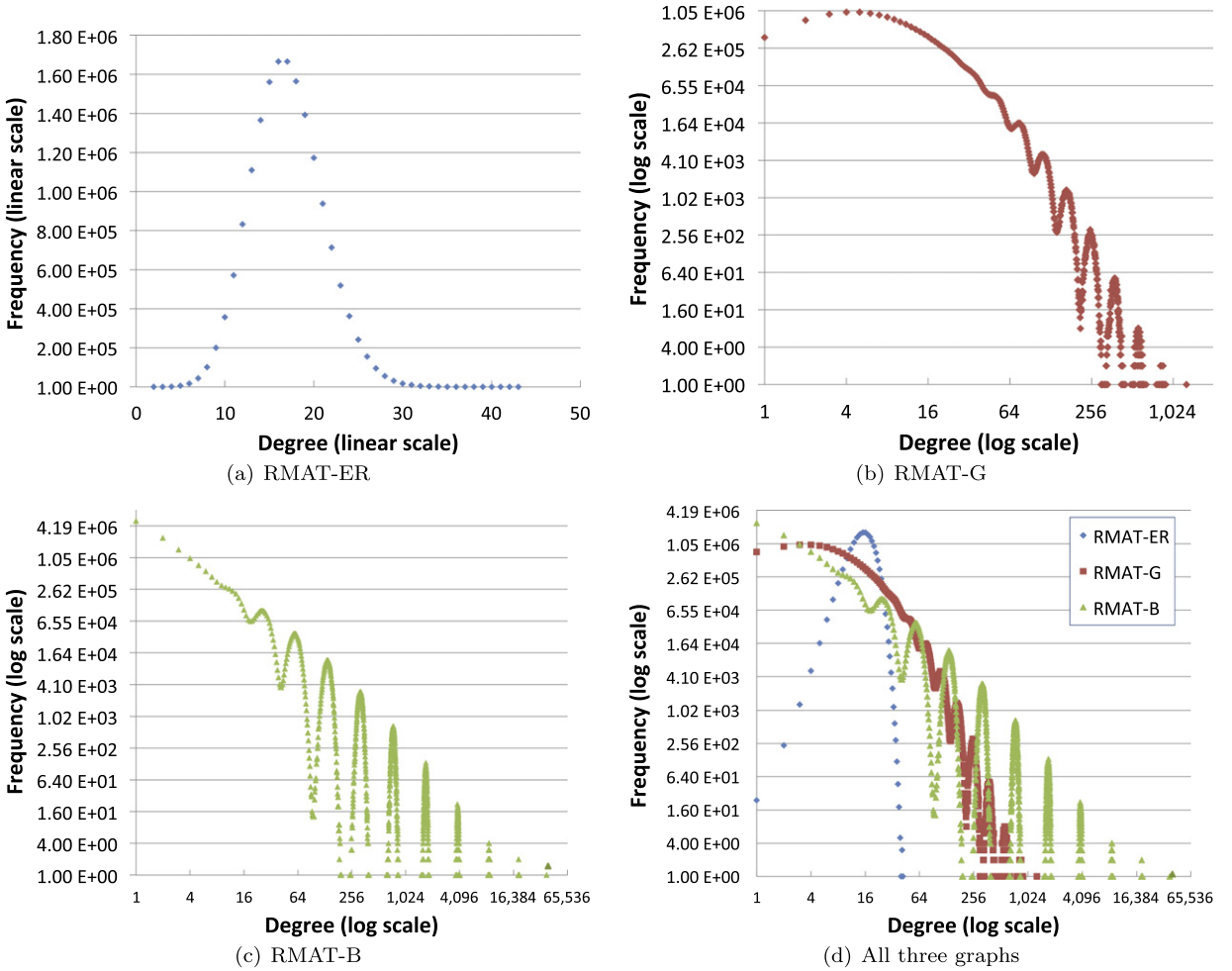
4.2. Characteristics of the test graphs

The test graphs are *designed* to represent instances posing varying levels of difficulty for the performance of the multithreaded coloring algorithms. The three graphs in Table 2 are nearly identical in size, implying that the serial work involved in the algorithms is the same, but they vary widely in degree distribution of the vertices and density of local subgraphs. Both of these have implications on available concurrency in algorithms Iterative and DataflowRecursive: large-degree vertices and dense subgraphs would increase the likelihood for conflicts to occur in Iterative and would cause more "serialization" in DataflowRecursive, thereby impacting parallel performance.

Fig. 4 summarizes the vertex degree distributions in the three graphs. The graph RMAT-ER belongs to the class of Erdős-Rényi random graphs, since $a=b=c=d=0.25$. Its degree distribution is expected to be *normal*. The single local maximum

Table 2
Structural properties of the test graphs.

Graph	No. vertices	No. edges	Avg. deg.	Max. deg.	Variance
RMAT-ER	16,777,216	134,217,654	16	42	16.01
RMAT-G	16,777,216	134,181,095	16	1278	415.72
RMAT-B	16,777,216	133,658,229	16	38,143	8,085.64



(a) RMAT-ER (b) RMAT-G (c) RMAT-B (d) All three graphs

Fig. 4. Degree distribution of the three test graphs listed in Table 2.

observed in the corresponding curve in the degree distribution plot depicted in Fig. 4(a) is consistent with this expectation. In the graphs RMAT-G and RMAT-B, "subcommunities" (dense local subgraphs) are expected to form because of the larger values of a and d compared to b and c; the subcommunities are coupled together in accordance with the values of b and c. The multiple local maxima observed in the degree distributions of these two graphs, shown in Fig. 4(b) and (c), correlate with the existence of such communities.

Besides degree distribution, the three test graphs also vary highly in terms of maximum degrees and degree variances (see Table 2). As an additional measure for the structural variation represented by the three graphs, we computed the clustering coefficients of vertices in these graphs using the tool GraphCT.[1] The *local clustering coefficient* of a vertex v is the ratio between the actual number of edges among the neighbors $adj(v)$ of v to the total possible number of edges among $adj(v)$ [33]. The average clustering coefficient in the graph is the sum of all local clustering coefficients of vertices divided by the number of vertices.

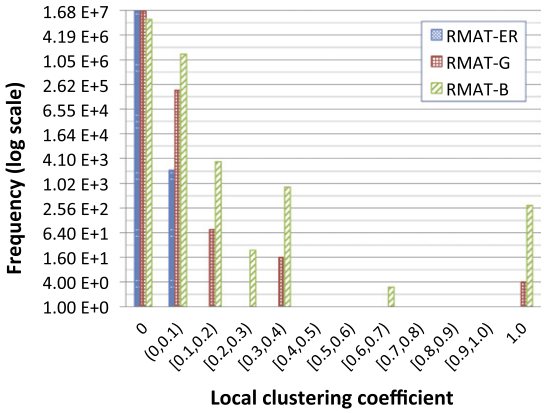
Tabulated in Fig. 5(a) is the average clustering coefficient in each of the three test graphs. The numbers there show orders of magnitude progression as one goes from RMAT-ER to RMAT-G to RMAT-B. Fig. 5(b) shows the distribution of local clustering coefficients, categorized in blocks of values. For the RMAT-ER graph, the local clustering coefficient of any vertex is found to be either 0, or between 0 and 0.1, whereas the values are spread over a wider range for the RMAT-G graph and even wider range for the RMAT-B graph. Larger local clustering coefficients in general indicate the presence of relatively dense local subgraphs, or subcommunities, as referred to earlier.

Finally, to visualize the difference in the structure represented by the three graphs, we include in Fig. 6 plots generated with MATLAB's `spy` function of the adjacency matrices of scaled-down versions of the three test graphs (1024 ver-

[1] Available at http://trac.research.cc.gatech.edu/graphs/wiki/GraphCT.

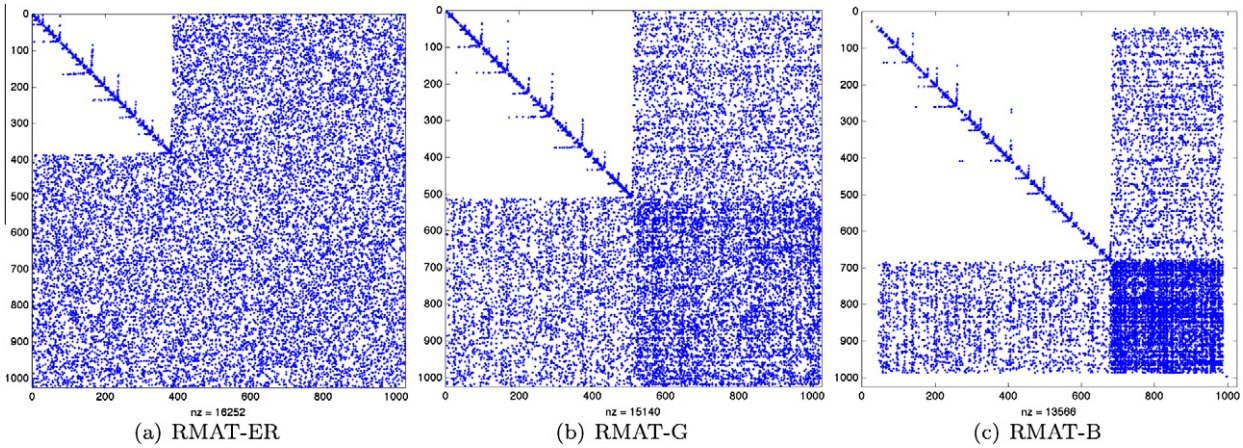
	Avg. CC
RMAT-ER	$1.00e-6$
RMAT-G	$1.20e-5$
RMAT-B	$3.43e-4$

(a) Average clustering coefficient



(b) Local clustering coefficient distribution

Fig. 5. Clustering coefficient data on the three test graphs listed in Table 2.



(a) RMAT-ER (b) RMAT-G (c) RMAT-B

Fig. 6. `Spy` plots of the adjacency matrices of scaled-down versions of the three graphs with the columns reordered using Approximate Minimum Degree to aid visual appeal. Each matrix has $2^{10} = 1024$ columns and 16,252 nonzeros.

tices and 16,252 edges). To aid visual appeal, in the `spy`-plots, the columns have been re-ordered using the Approximate Minimum Degree (AMD) routine in MATLAB; loosely speaking, AMD orders lower degree vertices (columns) before higher degree vertices in a graph model of sparse Gaussian elimination that updates the degrees of vertices dynamically [34].

Based on the variation in maximum degree value, degree distribution, and clustering coefficient values the three test graphs represent, one can see that a fairly wide spectrum of input types is covered in the experimental study.

5. Experimental results

With the test graphs discussed in Section 4 serving as inputs, we present in this section experimental results on the scalability of and the number of colors used by the algorithms Iterative and DataflowRecursive when they are run on the three

Table 3
Overview of compilers and flags used in the experiments.

	Compiler	Flag
Nehalem	Intel 11.1	-fast
Niagara 2	Sun Studio 5.9	-fast -xopenmp -m64
XMT	Cray Native C/C++	-par

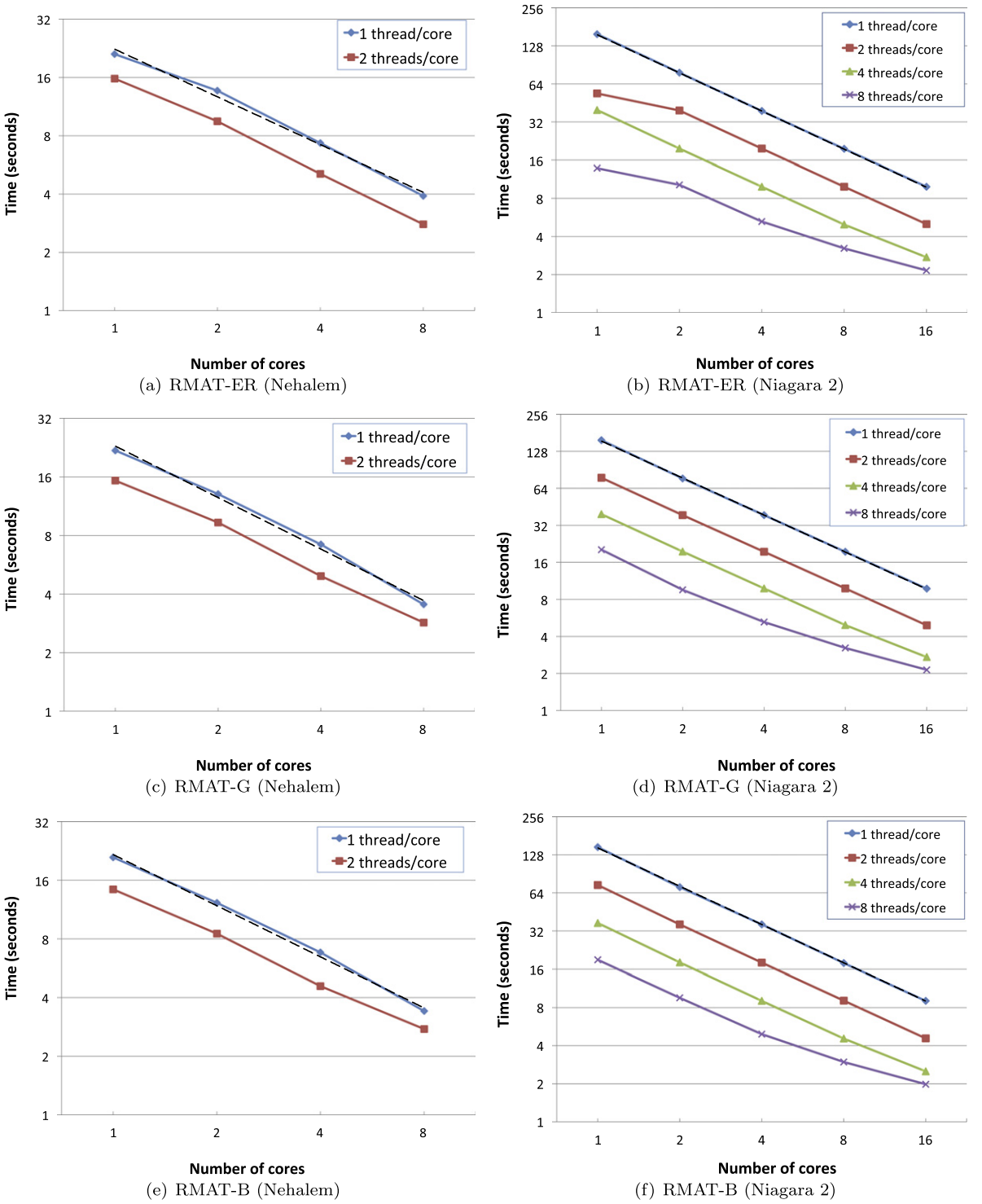


Fig. 7. Strong scaling of the ITERATIVE algorithm on Nehalem and Niagara 2 for the graphs listed in Table 2. On each platform, results for different number of threads per core are shown. The dashed-line curve in each chart corresponds to *ideal* speedup in the case where one thread per core is used. In all charts, both axes are in logarithmic scale.

platforms described in Section 2. We begin the section by discussing various matters on experimental setup in Sections 5.1–5.4. The scalability results are presented in Sections 5.5, 5.6, and 5.7, and the results on the number of colors used are presented in Section 5.8. The purpose of Section 5.5 is to discuss the scalability results on each of the three platforms *separately*

while that of Section 5.6 is to draw comparisons. Section 5.7 shows results analyzing the performance of Iterative in more detail.

5.1. Locality not exploited

The R-MAT algorithm tends to generate graphs where most high-degree vertices are of low indices. In the tests we run, we *randomly* shuffled vertex indices in order to reduce the benefits of this artifact on architectures with caches and to minimize memory-hotspotting. The vertices were then colored in the resulting numerical order in both algorithms.

5.2. Compilers used

Algorithm Iterative was implemented on the Nehalem and Niagara 2 platforms using OpenMP. On the Cray XMT, both Iterative and DataflowRecursive were implemented using the XMT programming model. Table 3 summarizes the compilers and flags used on each platform.

5.3. Thread scheduling

In both Iterative and DataflowRecursive, there is a degree of freedom in the choice of thread *scheduling policy* for parallel execution. Both the OpenMP and the XMT programming models offer directives supporting different kinds of policies. We experimented with various policies (static, dynamic, and guided with different block sizes) on the three platforms to assess the impact of the policies on performance. We observed no major difference in performance for the experimental setup we have, that is, RMAT-generated graphs with vertex indices randomly shuffled. Note, however, that for vertex orderings that exhibit some locality (e.g., some high degree vertices are clustered together) scheduling policies could affect performance due to cache effects. In the results reported in this paper, we have used the default *static* scheduling on all platforms.

5.4. Thread binding

In the experiments on scalability, we used various platform-specific thread binding mechanisms. On the Nehalem, binding was achieved using a *high-level affinity interface*, the environment variable `KMP_AFFINITY` from the Intel compiler. Recall that the Nehalem has two sockets with four cores per socket, and two threads per core, amounting to 16 threads in total. The eight threads on the four cores of the first socket are numbered (procID) as $\{(1,9),(3,11),(5,13),(7,15)\}$, where the first pair corresponds to the threads on the first core, the second pair to the threads on the second core, etc. Similarly the remaining eight threads on the second socket are numbered as $\{(0,8),(2,10),(4,12),(6,14)\}$. With such topology in place, we bound the OpenMP threads to the cores by assigning the procID to `KMP_AFFINITY`. For example, to run a job with four OpenMP threads on four cores (one thread per core), we set `KMP_AFFINITY="verbose,proclist=[9,8,15,14]"` and `OMP_NUM_THREADS = 4`.

Notice here that threads from the two sockets, rather than from only one, are engaged. Note also that the threads are bound to cores that are as far apart as possible. These seemingly counter-intuitive choices are made to maximize bandwidth. Recall that the memory banks in the Nehalem system (see Fig. 1) are associated with sockets, and each socket has independent channels to its memory bank. Engaging the two sockets would thus increase the effective memory bandwidth available. Such a choice could, however, be counter-productive for small problems that fit in a single bank, or for cache reuse between threads.

Table 4
Properties of the larger graphs used in the scalability study on the XMT.

Graph	Scale	No. vertices	No. edges	Max. deg.	Variance	% Isolated vertices
RMAT-ER	24	16,777,216	134,217,654	42	16.00	0
	25	33,554,432	268,435,385	41	16.00	0
	26	67,108,864	536,870,837	48	16.00	0
	27	134,217,728	1,073,741,753	43	16.00	0
RMAT-G	24	16,777,216	134,181,095	1,278	415.72	2.33
	25	33,554,432	268,385,483	1,489	441.99	2.56
	26	67,108,864	536,803,101	1,800	469.43	2.81
	27	134,217,728	1,073,650,024	2,160	497.88	3.06
RMAT-B	24	16,777,216	133,658,229	38,143	8,085.64	30.81
	25	33,554,432	267,592,474	54,974	9,539.17	32.34
	26	67,108,864	535,599,280	77,844	11,213.79	33.87
	27	134,217,728	1,071,833,624	111,702	13,165.52	35.37

Niagara 2 provides a mechanism for binding the threads to processors using `SUNW_MP_PROCBIND`. This was exploited in the experiments by binding 1, 2, 4, or 8 threads to each core to get results from 1 to 128 threads.

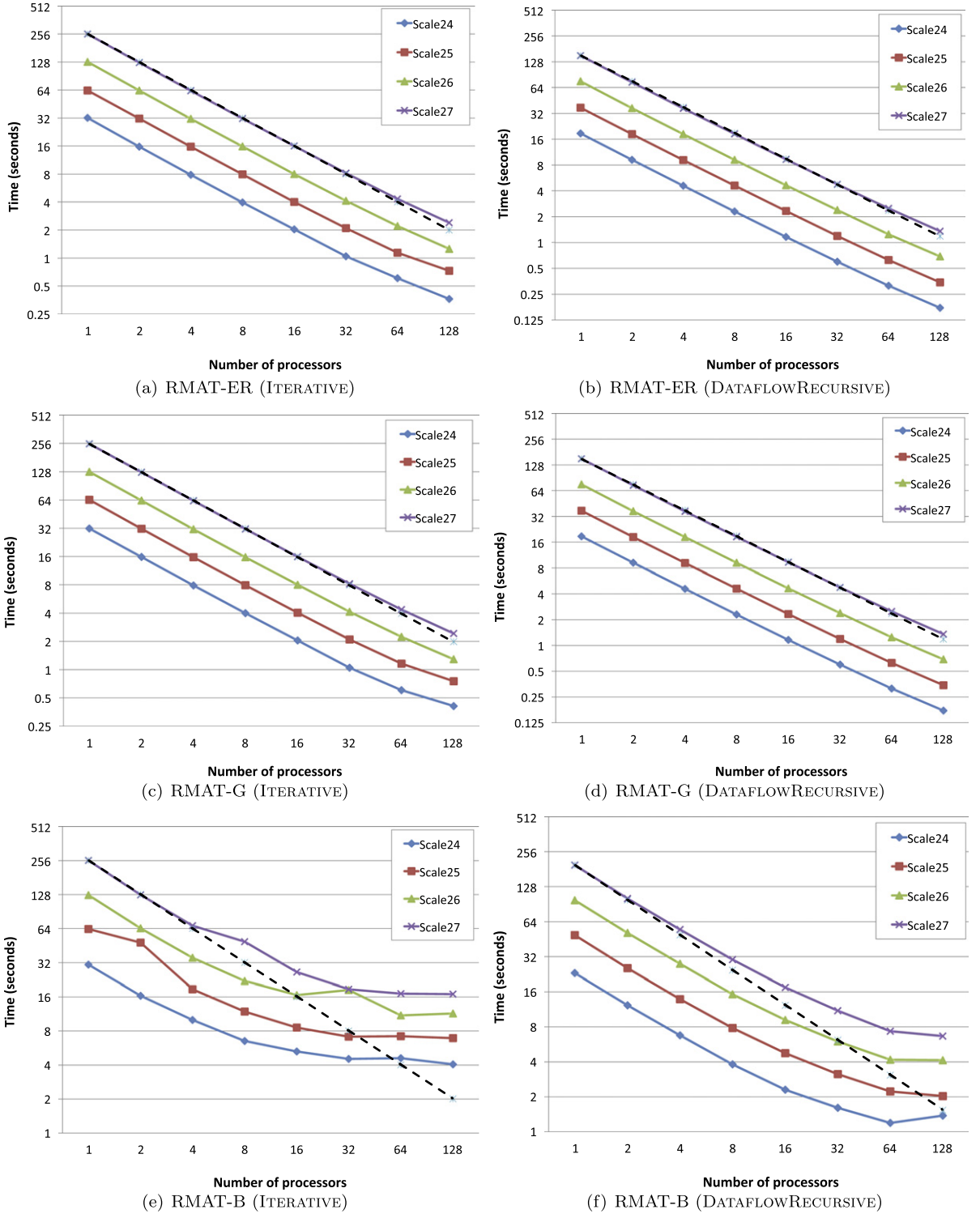


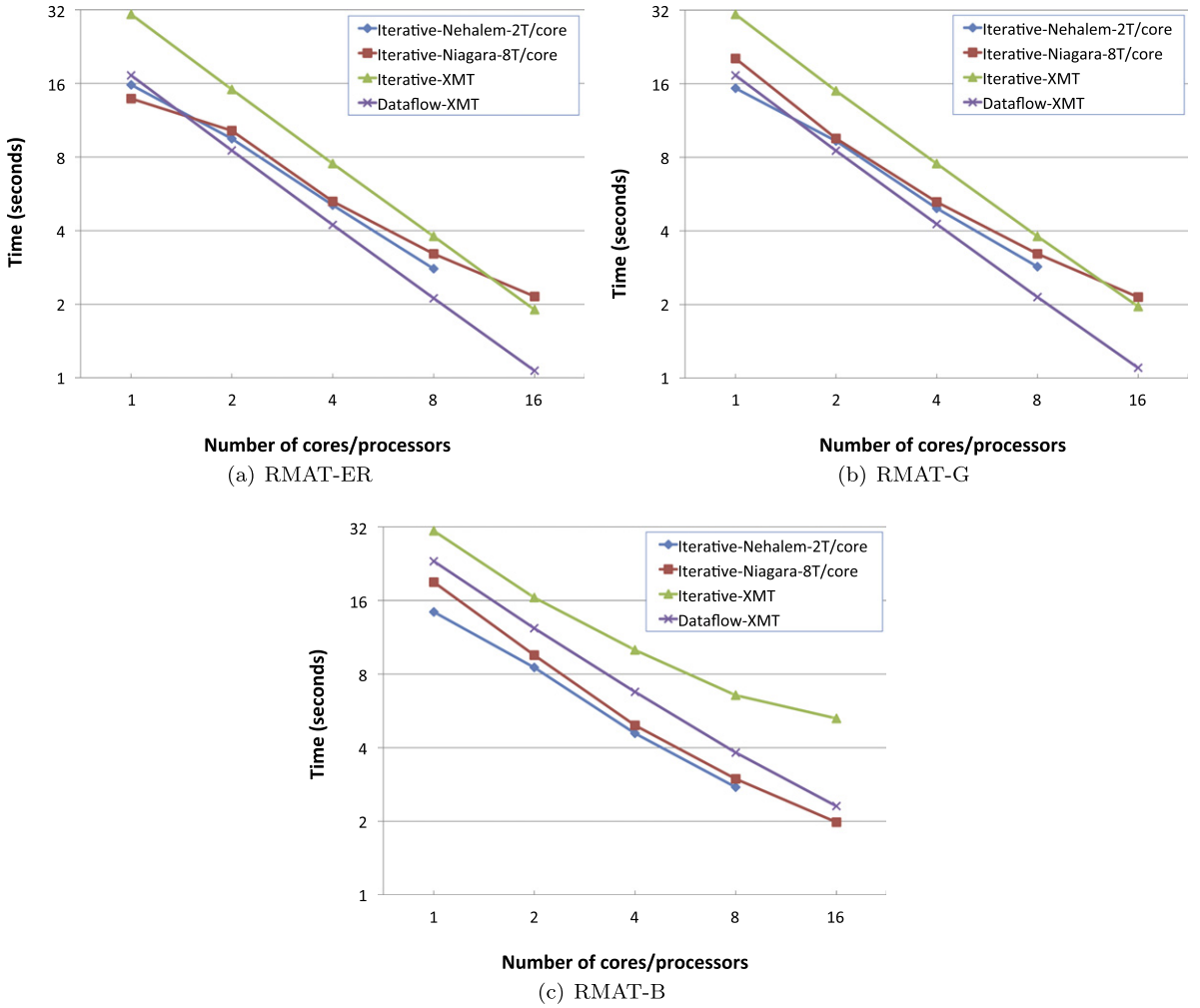
Fig. 8. Strong and weak scaling results on the Cray XMT for the graphs listed in Table 4. For each run, a system request of maximum 100 streams (threads) was placed. The dashed-line curve in each chart corresponds to *ideal* speedup on the largest graph (scale 27). In all charts, both axes are in logarithmic scale.

5.5. Scalability results

5.5.1. Scalability of Iterative on Nehalem and Niagara 2

Fig. 7 gives a summary of strong scaling results of algorithm Iterative on the three test graphs RMAT-ER, RMAT-G, and RMAT-B listed in Table 2. The left column shows results on the Nehalem platform, and the right column shows results on the Niagara 2 platform. On each platform, different numbers of threads per core were utilized via thread binding. In the sets of results on both architectures, the curve represented by the dashed line corresponds to *ideal* speedup in the case where one thread per core is used. It can be seen that the strong scaling behavior of algorithm Iterative is near-ideal on both architectures across all three input types considered, when one thread per core is used.

The results in Fig. 7 corresponding to the use of multiple threads per core on both the Nehalem and Niagara 2 show the benefit of exploiting simultaneous multithreading (SMT). On the Nehalem, by using two threads per core, instead of one, we obtained relative speedups ranging from 1.2 to 1.5, depending on the number of cores used and the type of input graph considered. The observed speedup due to SMT is less than the ideal factor of two, and the difference is more pronounced when the number of cores increases to eight. Note that the speedup is greater for the RMAT-ER graph than for the RMAT-B and RMAT-G graphs. This is because the memory accesses in the RMAT-ER graphs pose less performance challenges than the accesses in the RMAT-B and RMAT-G graphs, due to the presence of a larger number of dense subgraphs in the latter two. The speedup we obtained via the use of SMT on the Nehalem is consistent with the results reported in [35], where the authors measured the runtime reduction (speedup) obtained by employing SMT on the Nehalem on seven application codes. The authors found only a 50% reduction in the best case, and for four of the codes, they observed an increase in runtime (slowdown), instead of decrease (speedup).



(a) RMAT-ER (b) RMAT-G

(c) RMAT-B

Fig. 9. Strong scaling results on *all* three platforms for the graphs listed in Table 2. On the XMT, a system request was placed for maximum 100 streams (threads). In all charts, both axes are in logarithmic scale.

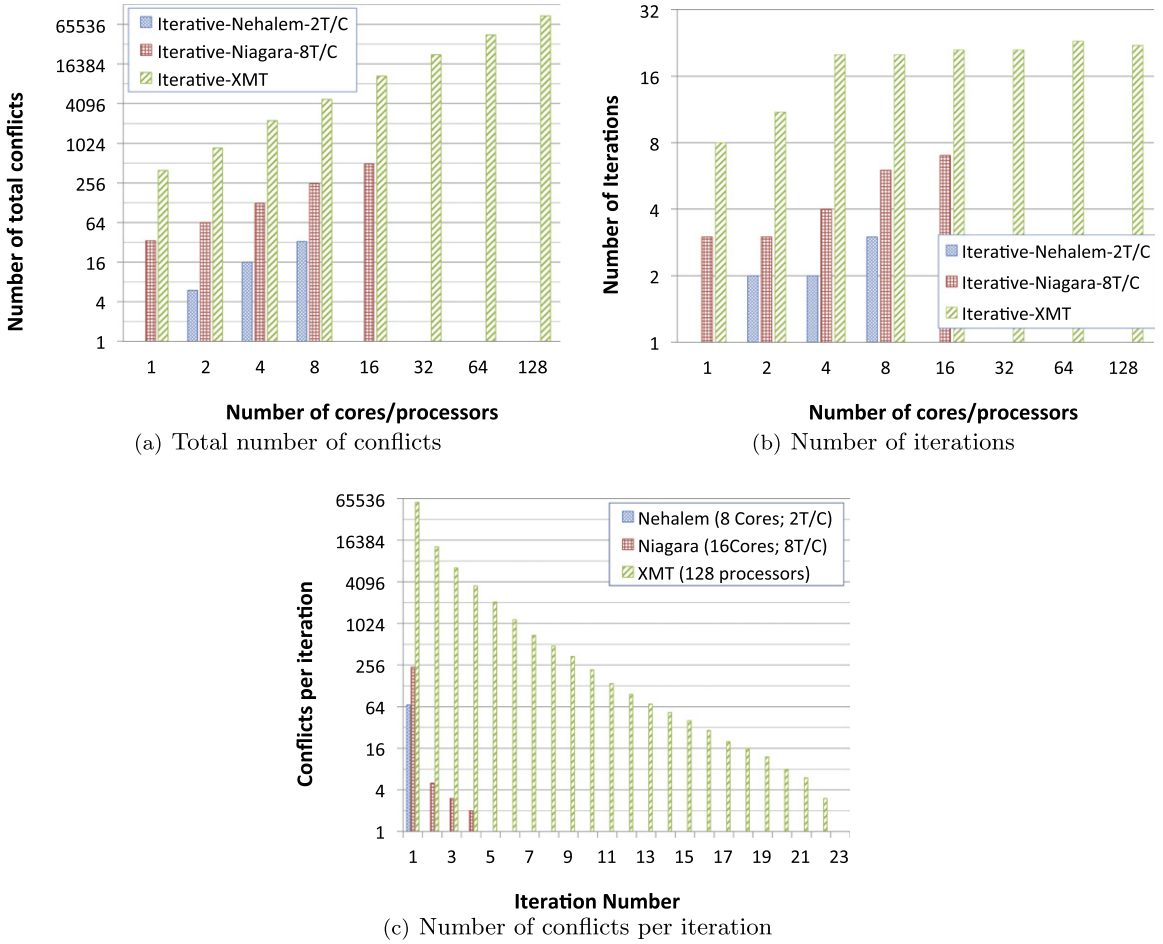
The speedup due to SMT we obtained on the Niagara 2 (the right column in Fig. 7) is more impressive than that on the Nehalem. By and large, we observed that the runtime of ITERATIVE using $2T$ threads on C cores is about the same as the runtime using T threads on $2C$ cores, except when T is 4, in which case the relative gain due to SMT is smaller.

5.5.2. Scalability of ITERATIVE and DATAFLOWRECURSIVE on the XMT

5.5.2.1. Larger test graphs. The Cray XMT provides about 1 TB of global shared memory and massive concurrency ($128 \times 128 = 16{,}384$-way interleaved multithreading). Both capabilities are much larger than what is available on the other two platforms. To make use of this huge resource and assess scalability, we experimented with larger RMAT graphs generated using the same parameters as in the experiments thus far but with different *scales*. The characteristics of these graphs are summarized in Table 4. In the table, the number of vertices (third column) is equal to 2^s, where s is the *scale* shown in the second column.

5.5.2.2. Thread allocation. In contrast to the Niagara 2 and Nehalem, where thread allocation and binding is *static*, the corresponding tasks on the XMT are *dynamic*. However, one can place a system request for a maximum number of *streams* (the XMT term for threads) for a given execution on the XMT. In our experiments, we requested 100 streams, an empirically determined optimal number for keeping all processors busy. The runtime system then decides as to how many will actually get allocated. We observed that at the beginning of the execution of the coloring algorithms, we do get close to 100 streams. Then, the number drastically decreases and gets to around 5 towards the end of the execution, when little computational work is left.

5.5.2.3. Scalability results. The scalability results we obtained on the XMT with the maximum 100 streams system request and using the large test graphs listed in Table 4 are summarized in Fig. 8; the left set of figures shows results on algorithm



(a) Total number of conflicts

(b) Number of iterations

(c) Number of conflicts per iteration

Fig. 10. (a) Comparison of total number of conflicts generated in the ITERATIVE algorithm on the different platforms for the RMAT-B graph listed in Table 2. (b) Total number of iterations the algorithm needed to complete on the same graph. (c) Breakdown of number of conflicts per iteration.

Iterative and the right set of figures shows results on algorithm DataflowRecursive. In each case the ideal (linear) scaling of the *largest* (scale 27) graph is depicted by the dashed line.

It can be seen that both Iterative and DataflowRecursive scale well to the maximum number of available processors (128) on the RMAT-ER and RMAT-G graphs; on the RMAT-B graph, DataflowRecursive scales to 64 processors and Iterative to 16 processors. The poorer scalability of Iterative is likely due to the relatively large number of conflicts generated (and consequently, the relatively large number of iterations required) as a result of the massive concurrency utilized on the XMT in coloring the vertices. It can also be seen that the runtime of Iterative is about twice that of DataflowRecursive. This is mainly because of the conflict-detection phase in Iterative, which entails a second pass through the (sub)graph data structure, a phase that is absent in DataflowRecursive.

Note that Fig. 8 also shows *weak* scaling results. In each subfigure, observe that the input size is doubled as one goes from one input size (say 24) to the next (say 25) and the number of processors is doubled as one goes from one data point on the horizontal axis to the next. Imagine running a horizontal line cutting through the four curves in each subfigure. In most of the figures such a line will intersect the four curves at points that correspond to a near-ideal weak scaling behavior.

5.6. Scalability comparison on the three architectures

Figs. 7 and 8 have already shown some of the differences in performance observed on the three architectures. For further cross-platform comparison, we provide in Fig. 9 a condensed summary of strong scaling results on all three platforms at once. The three subfigures show results corresponding to runtimes in seconds on the three test graphs of Table 2. In each subfigure four runtime plots are shown, three corresponding to the performance of Iterative on the three platforms and one corresponding to the performance of DataflowRecursive on the XMT.

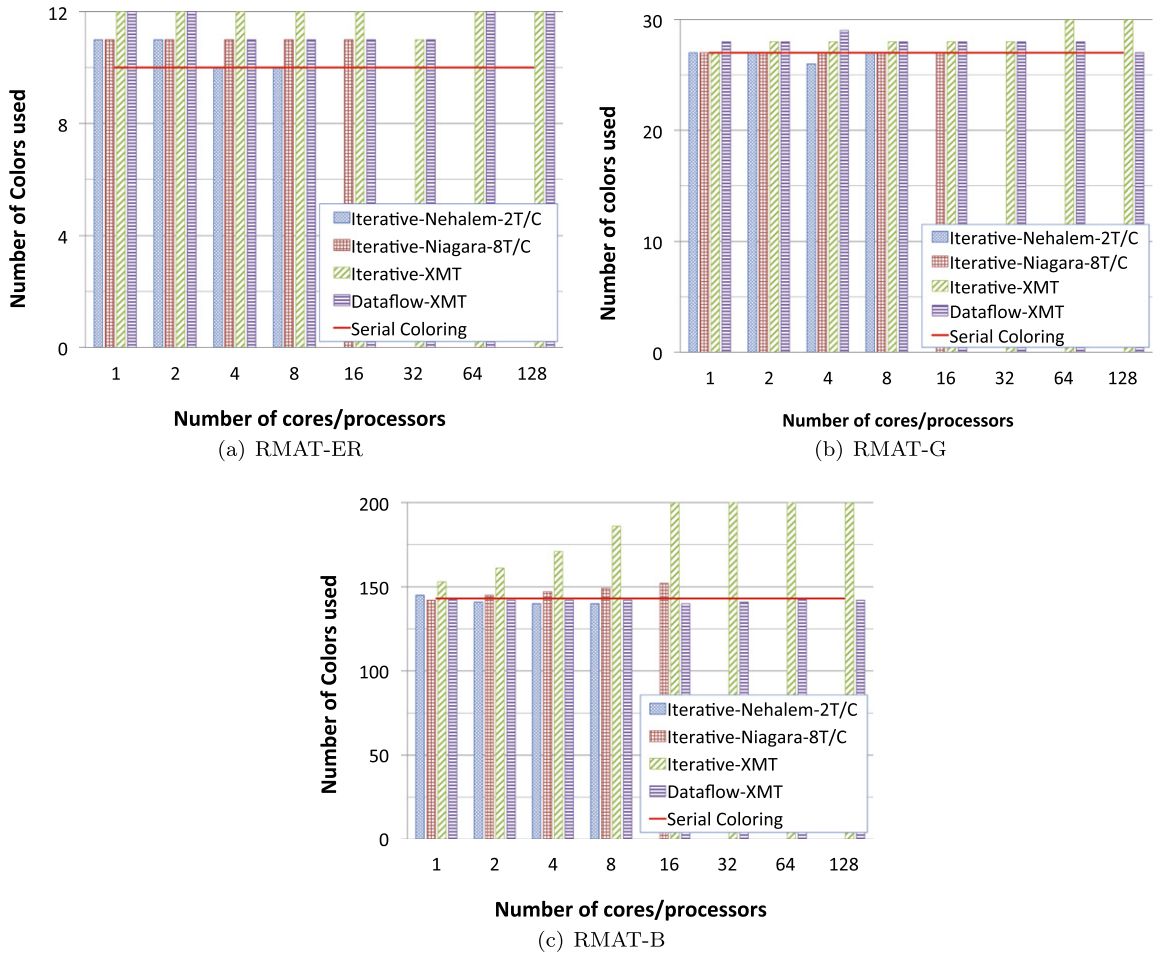


Fig. 11. Number of colors used by the two multithreaded algorithms on the various architectures for the graphs listed in Table 2. The number used by the *serial* greedy algorithm is depicted with the horizontal red line. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

To enable a common presentation on all three platforms, we have shown in Fig. 9 results on fewer processing units than what is available on the Cray XMT. In particular, we show results when up to 8 cores on the Nehalem, 16 cores on the Niagara 2, and 16 processors on the XMT are used. Note also that for a given number of cores (or processors in the case of XMT), the amount of concurrency utilized and the type of multithreading (MT) employed varies from platform to platform. On the XMT, out of the 128 threads available per processor, a system request for a maximum of 100 was placed. On the Niagara 2, the maximum eight threads available per core were used, and on Nehalem the two available threads per core were used. This means the *total* amount of concurrency involved is a possible maximum of 12,800 threads (with interleaved MT) on the XMT, exactly 128 threads (with simultaneous MT) on the Niagara 2, and exactly 16 threads (with simultaneous MT) on the Nehalem.

On the XMT, it can again be seen that DataflowRecursive scales nearly ideally on RMAT-ER and RMAT-G, and quite decently on RMAT-B, the most difficult input. Algorithm Iterative scales in an analogous fashion on the same platform. It can also be seen that Iterative scales well not only on the XMT, but also on the Niagara 2 and Nehalem platforms, in that relative ordering. The relative ordering is in part due to the difference in the amount of thread concurrency exploited, which is the highest on the XMT and the lowest on the Nehalem.

5.7. *Iterative* and degree of concurrency

For Iterative, higher concurrency would also mean higher likelihood for the occurrence of conflicts and, as a result, increase in the number of iterations involved. Fig. 10(a) shows the *total* number of conflicts generated during the course of algorithm iterative on the three platforms for the most hostile of the three input types, the RMAT-B graph. As expected, the relative ordering in terms of number of conflicts is Nehalem followed by Niagara 2 followed by the XMT. The number of these conflicts is seen to be small relative to the number of vertices (16 million) in the graph. As Fig. 10(b) shows, the total number of iterations the algorithm needed to resolve these conflicts is quite modest, even on the XMT, where the number of conflicts is relatively large. Fig. 10(c) shows how the total number of conflicts is divided across the iterations when 128 processors on the XMT, 16 cores of the Niagara 2 (with 8 threads per core) and 8 cores of the Nehalem (with 2 threads per core) are used. As this figure shows, about 90% of the conflicts occur in the first iteration and the number of conflicts drops drastically in subsequent iterations. This suggests that it might be worthwhile to switch to a sequential computation once a small enough number of conflicts is reached, rather than proceed iteratively in parallel. We will investigate this in future work.

5.8. Number of colors

The *serial* greedy coloring algorithm used 10, 27, and 143 colors on the three graphs RMAT-ER, RMAT-G, and RMAT-B listed in Table 2, respectively. These numbers are much smaller than the maximum degrees (42; 1,278; and 38,143) in these graphs, which are upper bounds on the number of colors the greedy algorithm uses. Clearly, the gap between the upper bounds and the actual number of colors used is very large, which suggests that the serial greedy algorithm is providing reasonably good quality solutions. Indeed experiments have shown that the greedy algorithm often uses near-optimal numbers of colors on many classes of graphs [9,13].

Fig. 11 summarizes the number of colors the multithreaded algorithms Iterative and DataflowRecursive use on the three test graphs as the number of cores/processors on the various platforms is varied. It can be seen that both algorithms use about the same number of colors as the serial greedy algorithm and the increase in number of colors with an increase in concurrency is none to modest—only for Iterative on the XMT (where concurrency is massive) and the graph RMAT-B do we see a modest increase in number of colors as the number of processors is increased.

6. Conclusion

We presented a heuristic multithreaded algorithm for graph coloring that is suitable for any shared-memory system, including multi-core platforms. The key ideas in the design of the algorithm are *speculation* and *iteration*. We also presented a massively multithreaded algorithm for coloring designed using *dataflow* principles exploiting the fine-grain, hardware-supported synchronization mechanisms available on the Cray XMT. Using a carefully chosen set of input graphs, covering a wide range of problem types, we evaluated the performance of the algorithms on three different platforms—Intel Nehalem, Sun Niagara 2, and Cray XMT—that feature varying degrees of multithreading and caching capabilities to tolerate latency. The iterative algorithm (across all three architectures) and the dataflow algorithm (on the Cray XMT) achieved near-linear speedup on two of the three input graph classes considered and moderate speedup on the third, most difficult, input type. These runtime performances were achieved without compromising the quality of the solution (i.e., the number of colors) produced by the underlying serial algorithm. We also characterized the input graphs and provided insight on bottlenecks for performance.

Backed by experimental results, some of the more general conclusions we draw about parallel graph algorithms include:

- simultaneous multithreading provides an effective way to tolerate latency, as can be seen from the experiments on the Nehalem and Niagara 2 platforms where the runtime using N threads on one core is found to be similar to the runtime using one thread on N cores,

- the impact of lower clock frequency and smaller cache memories can be ameliorated with a greater thread concurrency, as can be seen in the better performance obtained on the XMT and the Niagara 2 relative to the Nehalem,
- when supported by light-weight synchronization mechanisms in hardware, parallelism should be exploited at fine grain, and
- graph *structure* critically influences the performance of parallel graph algorithms.

We expect these insights to be helpful in the design of high performance algorithms for irregular problems on the impending many-core architectures.

Acknowledgments

We thank the anonymous referees and Fredrik Manne for their valuable comments on an earlier version of the manuscript. This research was supported by the U.S. Department of Energy through the CSCAPES Institute (Grants DE-FC02-08ER25864 and DE-FC02-06ER2775), by the National Science Foundation through Grants CCF-0830645, CNS-0643969, OCI-0904809, and OCI-0904802, and by the Center for Adaptive Supercomputing Software (CASS) at the Pacific Northwest National Laboratory. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under Contract DE-AC06-76L01830.

References

[1] A. Lumsdaine, D. Gregor, B. Hendrickson, J.W. Berry, Challenges in parallel graph processing, Parallel Processing Letters 17 (1) (2007) 5–20.
[2] J. Kurzak, D.A. Bader, J. Dongarra, Scientific Computing with Multicore and Accelerators, Computational Science Series, Chapman and Hall, CRC Press, 2011.
[3] D.A. Bader, G. Cong, Efficient graph algorithms for multicore and multiprocessors, in: S. Rajasekaran, J. Reif (Eds.), Handbook of Parallel Computing, Chapman and Hall/CRC Press, 2008, pp. 1–44 (Chapter 26).
[4] K. Madduri, D.A. Bader, J.W. Berry, J.R. Crobak, B. Hendrickson, Multithreaded algorithms for processing massive graphs, in: D.A. Bader (Ed.), Petascale Computing: Algorithms and Applications, Chapman and Hall/CRC Press, 2008, pp. 237–258.
[5] M. Jones, P. Plassmann, Scalable iterative solution of sparse linear systems, Parallel Computing 20 (5) (1994) 753–773.
[6] Y. Saad, ILUM: a multi-elimination ILU preconditioner for general sparse matrices, SIAM Journal on Scientific Computing 17 (1996) 830–847.
[7] D. Hysom, A. Pothen, A scalable parallel algorithm for incomplete factor preconditioning, SIAM Journal on Scientific Computing 22 (2001) 2194–2215.
[8] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, P. Hanrahan, Liszt: a domain specific language for building portable mesh-based PDE solvers, Supercomputing 3 (2011) 4.
[9] T.F. Coleman, J.J. Moré, Estimation of sparse Jacobian matrices and graph coloring problems, SIAM Journal on Numerical Analysis 20 (1) (1983) 187–209.
[10] T.F. Coleman, J.J. Moré, Estimation of sparse Hessian matrices and graph coloring problems, Mathematical Programming 28 (1984) 243–270.
[11] A.H. Gebremedhin, F. Manne, A. Pothen, What color is your Jacobian? Graph coloring for computing derivatives, SIAM Review 47 (4) (2005) 629–705.
[12] D. Zuckerman, Linear degree extractors and the inapproximability of max clique and chromatic number, Theory of Computing 3 (2007) 103–128.
[13] A.H. Gebremedhin, D. Nguyen, M.M.A. Patwary, A. Pothen, ColPack: software for graph coloring and related problems in scientific computing, ACM Transactions on Methametical Software, submitted for publication.
[14] D. Bozdağ, A.H. Gebremedhin, F. Manne, E.G. Boman, Ü.V. Çatalyürek, A framework for scalable greedy coloring on distributed-memory parallel computers, Journal of Parallel and Distributed Computing 68 (4) (2008) 515–535.
[15] D. Bozdağ, Ü.V. Çatalyürek, A.H. Gebremedhin, F. Manne, E.G. Boman, F. Özgünner, Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation, SIAM Journal on Scientific Computing 32 (4) (2010) 2418–2446.
[16] COLOR02/03/04: graph coloring and its generalizations. <http://mat.gsia.cmu.edu/COLOR03/>.
[17] DIMACS implementation challenges. <http://dimacs.rutgers.edu/Challenges/>
[18] J.S. Turner, Almost all k-colorable graphs are easy to color, Journal of Algorithms 9 (1) (1988) 63–82.
[19] D. Brélaz, New methods to color the vertices of a graph, Communications of the ACM 22 (4) (1979) 251–256.
[20] A. Coja-Oghlan, M. Krivelevich, D. Vilenchik, Why almost all k-colorable graphs are easy to color, Theory of Computing Systems 46 (2010) 523–565.
[21] A.H. Gebremedhin, A. Tarafdar, F. Manne, A. Pothen, New acyclic and star coloring algorithms with application to computing Hessians, SIAM Journal on Scientific Computing 29 (2007) 1042–1072.
[22] A.H. Gebremedhin, A. Pothen, A. Tarafdar, A. Walther, Efficient computation of sparse Hessians using coloring and automatic differentiation, INFORMS Journal on Computing 21 (2) (2009) 209–223.
[23] D. Molka, D. Hackenberg, R. Schone, M.S. Muller, Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system, in: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, Washington, DC, USA, 2009, pp. 261–270.
[24] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sana, D. Sheahan, L. Spracklen, A. Wynn, UltraSPARC T2: a highly-threaded, power-efficient SPARC SOC, in: Solid-State Circuits Conference, 2007, ASSCC '07, IEEE Asian, 2007, pp. 22–25.
[25] J. Feo, D. Harper, S. Kahan, P. Konecny, Eldorado, in: "Eldorado", Proceedings of the 2nd Conference on Computing Frontiers, ACM, New York, NY, USA, 2005, pp. 28–34.
[26] J. Nieplocha, A. Márquez, J. Feo, D. Chavarría-Miranda, G. Chin, C. Scherrer, N. Beagley, Evaluating the potential of multithreaded platforms for irregular scientific computations, in: Proceedings of the 4th Conference on Computing Frontiers, ACM, New York, NY, USA, 2007, pp. 47–58.
[27] M. Luby, A simple parallel algorithm for the maximal independent set problem, SIAM Journal on Computing 15 (4) (1986) 1036–1053.
[28] M.T. Jones, P. Plassmann, A parallel graph coloring heuristic, SIAM Journal on Scientific Computing 14 (3) (1993) 654–669.
[29] I. Finocchi, A. Panconesi, R. Silvestri, Experimental analysis of simple, distributed vertex coloring algorithms, Algorithmica 41 (1) (2004) 1–23.
[30] A.H. Gebremedhin, F. Manne, Scalable parallel graph coloring algorithms, Concurrency: Practice and Experience 12 (2000) 1131–1146.
[31] D. Chakrabarti, C. Faloutsos, Graph mining: laws, generators, and algorithms, ACM Computing Surveys 38 (1) (2006) 2.
[32] D. Bader, J. Feo, J. Gilbert, J. Kepner, D. Keoster, E. Loh, K. Madduri, B. Mann, T. Meuse, HPCS scalable synthetic compact applications 2, 2007. <http://www.graphanalysis.org/benchmark/HPCS-SSCA2Graph-Theoryv2.1.doc>
[33] D.J. Watts, S.H. Strogatz, Collective dynamics of small-world networks, Nature 393 (6684) (1998) 440–442.
[34] P.R. Amestoy, T.A. Davis, I.S. Duff, An approximate minimum degree ordering algorithm, SIAM Journal on Matrix Analysis and Applications 17 (4) (1996) 886–905.
[35] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, J.C. Sancho, A performance evaluation of the Nehalem quad-core processor for scientific computing, Parallel Processing Letters 18 (4) (2008) 453–469.