Distributed-Memory Parallel Algorithms for Matching and Coloring

Ümit V. Çatalyürek*, Florin Dobrian†, Assefaw Gebremedhin‡, Mahantesh Halappanavar§, Alex Pothen‡
* *Depts. of Biomedial Informatics and Electrical & Computer Engineering, The Ohio State University*
Email: umit@bmi.osu.edu
† *Conviva*
Email: dobrian@conviva.com
‡ *Department of Computer Science, Purdue University*
Email: {agebreme,apothen}@purdue.edu
§ *Pacific Northwest National Laboratory*
Email: Mahantesh.Halappanavar@pnl.gov

Abstract

We discuss the design and implementation of new highly-scalable distributed-memory parallel algorithms for two prototypical graph problems, edge-weighted matching and distance-1 vertex coloring. Graph algorithms in general have low concurrency, poor data locality, and high ratio of data access to computation costs, making it challenging to achieve scalability on massively parallel machines. We overcome this challenge by employing a variety of techniques, including speculation and iteration, optimized communication, and randomization. We present preliminary results on weak and strong scalability studies conducted on an IBM Blue Gene/P machine employing up to tens of thousands of processors. The results show that the algorithms hold strong potential for computing at petascale.

Key words: Combinatorial scientific computing; graph coloring; graph matching; parallel algorithms; petascale computing; distributed-memory architectures; MPI

1. Introduction

We discuss on-going work on the design and implementation of new scalable distributed-memory parallel algorithms for two prototypical combinatorial problems of central importance to scientific computing, edge-weighted matching and distance-1 vertex coloring, and present preliminary results from experiments conducted on an IBM Blue Gene/P employing up to tens of thousands of processors.

Developing scalable parallel graph algorithms on tera-scale (and peta-scale) distributed-memory architectures is challenging for several reasons [17]. Many graph algorithms are inherently serial in nature, and therefore require nontrivial algorithmic techniques for creating concurrency. Graph algorithms possess poor data locality making it difficult to obtain good memory system performance on distributed-memory architectures. Computation and communication schedules can be determined often only at runtime, which makes compile-time prefetching techniques inapplicable. There is relatively low computation per communicated word, causing loss of performance on distributed-memory architectures where communicating a word typically costs much more than computing with it.

We overcome many of these challenges for the two problems we consider by employing a variety of techniques.

An optimal solution for the edge-weighted matching problem can be obtained in polynomial time on a serial machine. However, such optimal algorithms are ill-suited for parallelization since they involve long augmenting paths. We circumvent this difficulty by working with a suitable *approximation* algorithm instead. The approximation algorithm we work with has a near linear-time complexity, guarantees a solution that is at least half of the optimal weight, and is amenable for parallelization as it is based on the idea of identifying *locally dominant* (heaviest) edges. Although the algorithm guarantees a half-approximate solution, experimental evidence indicates that the algorithm often gives higher than 90% of the optimal weight for graphs that arise in practice. See Table 1.1 for sample results.

Matrix	#Vertices	#Edges	Quality
ASIC_680k	$1,365,724$	$3,871,773$	99.99%
Hamrle3	$2,894,720$	$5,514,242$	99.36%
rajat31	$9,380,004$	$20,316,253$	100.00%
cage14	$3,011,570$	$27,130,349$	100.00%
ldoor	$1,904,406$	$46,522,475$	100.00%
audikw_1	$1,887,390$	$77,651,847$	100.00%

Table 1.1: *Quality of the solution computed by the half-approximation matching algorithm.* The data set consists of the bipartite graph representation of randomly chosen matrices from the University of Florida Sparse Matrix Collection. The first column identifies the matrices, the second and third columns provide the size of the corresponding graphs, and the fourth column provides the quality of the suboptimal solutions relative to optimal solutions (percentages).

The distance-1 vertex coloring problem is NP-hard to solve optimally [6]. Yet, a *greedy* algorithm, which runs in linear time in the number of edges and uses at most

$\Delta + 1$ colors, where Δ is the maximum degree in the graph, often yields near-optimal solution for graphs that arise in practice when good *vertex ordering* techniques are employed—the near optimality of the solutions can be verified by computing appropriate lower bounds [8]. Such a greedy coloring algorithm is, however, sequential in nature. We use *speculation and iteration*, where concurrency is maximized by tentatively allowing inconsistencies and resolving them later (iteratively), as a technique for overcoming this obstacle. This iterative approach is further enhanced by using *randomization* in the conflict-resolution phase, where bias and computational load imbalance among processors is avoided by picking the vertex to be re-colored in the event of a conflict (an edge whose endpoints have received the same color) randomly rather than deterministically.

The input graph in both the matching and coloring algorithms is assumed to be already distributed among processors. In both algorithms, we exploit the parallelism created due to the initial data distribution. Furthermore, the cost associated with unavoidable communication is *optimized*. The optimization is achieved by aggregating frequent, small messages into infrequent, large messages, and by doing as much computation as possible in between communication phases, even if the computation has to be performed with incomplete information.

Employing the ingredients outlined above, we implemented the parallel algorithms for the matching and coloring problems using MPI. We show that the implementations scale well to tens of thousands of processors for graphs that are distributed on processors in such a way that the percentage of boundary vertices is low.

Our work is motivated by the enabling role graph matching and coloring (in many variations) play in computational science and engineering (CSE) and in high-performance computing. Examples of contexts in which some variant of graph matching is used to devise an effective solution strategy include: maximizing diagonal dominance in sparse linear solvers—to improve numerical stability in direct solvers and convergence rate in iterative solvers [4], [5]; computation of block triangular decomposition of sparse matrices [22]; computation of sparse bases for underdetermined matrices [21]; the coarsening phase of multilevel algorithms for graph partitioning [13]; discovery (or when specified, identification) of substructures in networks arising in bioinformatics [1], [16] and web technology applications [19]; etc. Likewise, a few examples of scenarios in which some variant of graph coloring is used to make overall computation efficient (or feasible) include: task scheduling and concurrency discovery in parallel computing [12], [24]; efficient computation of sparse Jacobian and Hessian matrices in numerical optimization [7]; frequency assignment in wireless networks [15]; etc.

There is a large body of work in the literature on graph matching and on coloring, but much of it is theoretical in nature. Few studies are concerned with the development of efficient implementations useful for CSE. And there does not seem to exist prior work on parallel implementations suitable for computing at the tera-scale and beyond. In this regard, our work stands in contrast to other efforts. Manne and Bisseling [18] had described, albeit briefly, a variant of the parallel half-approximation algorithm for edge-weighted matching presented here and reported results using up to 32 processors for two test cases. We achieve much greater scalability primarily by paying careful attention to the communication cost involved. A significant part of the improvement comes from aggressive message bundling, where messages sent between the same pair of processors are grouped as often as possible.

The parallel coloring algorithm presented here is derived from the framework for parallelizing greedy coloring (based on speculation and iteration) developed by Bozdag *et al.* [3]. The work in [3] showed that algorithms based on speculation and iteration outperform previously known algorithms that rely on iterative computation of *maximal independent sets* in parallel. The work also showed how specialized algorithms—tailored to specific input structures and computational scenarios—can be designed using the framework. Experiments in [3] showed that the best specialized algorithm scaled well up to a few hundred processors. The algorithm presented here extends the scalability range to several thousands of processors. The improvement, here again, is due to the use of new techniques for reducing interprocessor communication cost.

2. Preliminaries

Graph matching and coloring problems come in many variations depending on whether the problem is defined on a bipartite or a non-bipartite graph, on whether or not weights are associated with the vertices and/or edges of a graph, and on the specifics of the computational scenario under consideration [7], [9]. We focus in this paper on two specific variants, distance-1 vertex coloring and edge-weighted matching on non-bipartite graphs.

The goal of the *distance-1 vertex coloring* problem is to assign colors to the vertices of a graph such that every pair of adjacent vertices receives different colors and the number of colors used is minimized. In the *edge-weighted matching* problem, the goal is to find a subset M of the edges in an edge-weighted graph such that every vertex in the graph is incident on at most one edge in M and the sum of the weights of the edges in M is maximized. As mentioned earlier, the distance-1 coloring problem in NP-hard, whereas the edge-weighted matching problem is polynomial time solvable. We rely here on a linear-time greedy heuristic for the coloring problem and near linear-time half-approximation algorithm for the matching problem.

In the parallel versions of these algorithms that we describe in the next two sections, the input graph is assumed to be *partitioned* and distributed among the available processors in some reasonable way. By partitioning is meant that processors are assigned nearly the same number of vertices and the number of *cross edges*—edges that connect vertices assigned to different processors—is relatively small. The partitioning classifies the vertices as *interior* and *boundary*. An interior vertex is a vertex all of whose neighbors are assigned to the same processor as itself whereas a boundary vertex is a vertex that has at least one neighbor assigned to a different processor. We say that a processor *owns* the vertices assigned to it, and assume that a processor is responsible for coloring (or matching) the vertices it owns. We refer to an edge whose endpoints are assigned to the same processor as *interior*.

Clearly, the subgraphs induced by interior vertices (consisting of only interior edges) are independent of each other. Hence, a coloring in each can be computed independently of any other. In the coloring algorithm presented in this paper, these sub-solutions are concurrently computed and are conceptually pieced-together to give a partial solution to the original problem on the entire input graph. To obtain a complete solution, a coloring of the remainder of the graph, the *interface graph*, needs to be computed in parallel—and this is the phase in our algorithm where speculation is used and techniques for minimizing inter-processor communication cost are employed.

In the parallel matching algorithm, the computation on the subgraphs and the interface graph is not as cleanly delineated as in the parallel coloring algorithm. Dependencies between edges make it necessary to interleave the processing of interior and cross edges. On each processor, the computation considers interior edges for matching at first. Then, processors exchange messages as cross edges are considered for matching. This process is alternated in an asynchronous manner until the algorithm terminates.

3. Parallel Matching

The approximation algorithm we use for the parallel computation of a matching M in a graph $G = (V, E)$ is based on the concept of *local dominance*: edges that are heavier than their neighbors are moved from E to M (M is initially empty) and their neighbors (free edges) are removed from E; the process continues until E becomes empty. Preis [23] showed that such a procedure guarantees a $1/2$ approximation factor. Describing the parallel matching algorithm based on this procedure in full detail requires more space than is available here. Hence we describe in adequate detail (i) the sequential algorithm and (ii) the communication involved in its parallel version within the simplified context of one vertex per processor. Then, we only sketch the details of the parallel algorithm in the practical context when the number of vertices per processor is much larger than one.

3.1. The Sequential Algorithm

A matching based on local dominance can be computed in linear ($O(|E|)$) time [23]. Unfortunately, such an implementation is not well suited for parallelization, since it requires inspecting augmenting paths that can be arbitrarily long.

Hoepman [10] and later Manne and Bisseling [18] discuss an implementation, based on maintaining *candidate mates*, that is better suited for parallelization. The complexity of a basic variant of this algorithm is $O(|E|\Delta)$, where Δ is the maximum vertex degree. The run time can be improved to $O(|E| \log \Delta)$ by sorting the adjacency list of each vertex based on edge-weights. However, if the weights are distributed uniformly at random then the expected run time is $O(|E|)$ [18].

In the candidate mate-based algorithm that we use, for every vertex v, a candidate mate, denoted as $candidateMate(v)$, is maintained. This becomes the true mate, denoted as $mate(v)$, if v gets matched to it ($mate(v)$ is set to 0 otherwise). Initially, for every vertex v, $candidateMate(v)$ is set to be the other endpoint of a heaviest edge incident on v. If there is more than one such edge then ties are broken by choosing the neighbor with the smallest label. In addition to the candidate mate, for every vertex v, a set $S(v)$ that keeps track of all remaining edges incident on v is maintained. Each set $S(v)$ is initialized to $adj(v)$.

The algorithm then proceeds by identifying locally dominant edges, i.e., an edge (u, v) such that $candidateMate(u) = v$ and $candidateMate(v) = u$. Each such edge is moved from E to M and the remaining edges incident on vertices u and v are removed from E, a task accomplished by setting $mate(u) = v$, $mate(v) = u$, $S(u) = \emptyset$ and $S(v) = \emptyset$, and by removing u and v from $S(w)$, for every vertex w adjacent to u or v. Also, for each such neighbor vertex w, if $candidateMate(w)$ happens to be equal to u or v, it is necessary to recompute its value, considering the endpoints of the remaining edges incident on w, which are stored in $S(w)$. When no edges remain in $S(w)$, $candidateMate(w)$ becomes 0.

In a sequential computation this can be implemented efficiently by processing the matched vertices through a queue Q. The termination can be controlled either by the status of E or Q, since both become empty at the same time. Each time an edge (u, v) is added to M, its two endpoints u and v are added to Q. When a vertex is removed from Q, for each one of its exposed neighbors w, $S(w)$ and $candidateMate(w)$ are updated accordingly.

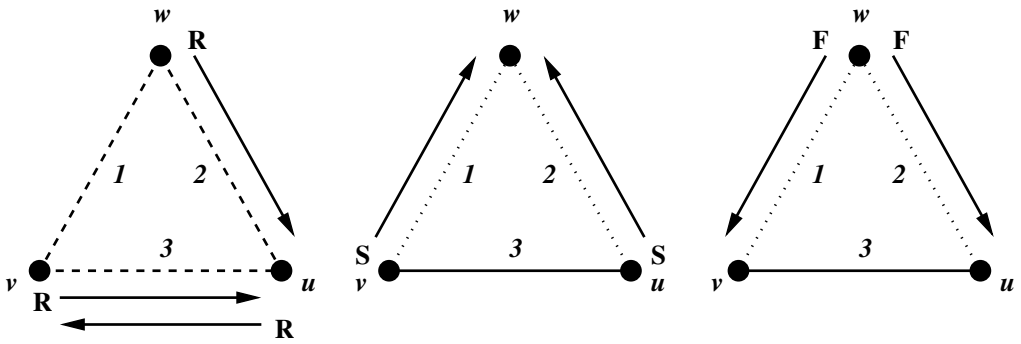

Figure 3.1: Visualization of the basic communication in the parallel matching algorithm, using a simple graph (three different stages of the computation are illustrated).

3.2. A Simple Parallel Algorithm

We now consider how the candidate mate-based algorithm just described can be parallelized in the simple case where there is only one vertex mapped to a processor. (The procedure we describe is similar to the procedure presented in [10] and [18]).

Three message types are used to communicate between processors:

- `REQUEST`—to signal a matching preference,
- `SUCCEEDED`—to signal that a vertex has already been successfully matched and is therefore no longer available for matching, and
- `FAILED`—to signal that a vertex cannot be matched at all.

Each such message is sent across some edge (u, v), from $p(u)$, the processor that owns the vertex u, to $p(v)$, the processor that owns the vertex v. Accordingly, each message contains the identities of the endpoints of the corresponding edges. All communication is asynchronous and point-to-point.

The computation begins by initializing $candidateMate(v)$ and $S(v)$, for every vertex v, as before (the processor that owns vertex v stores all the information about v, including $adj(v)$), and by sending $\texttt{REQUEST}(v, candidateMate(v))$ messages. After that, incoming messages are processed: processor $p(v)$ listens as long as there still exist edges incident on v (determined through the status of $S(v)$), and its further actions involve sending one of the three types of messages. Message processing is guaranteed to remove edges until every set $S(v)$ becomes empty (thus E becomes empty as well), therefore the computation is guaranteed to terminate.

For every vertex v an additional set $R(v)$ is used in the parallel implementation, in order to keep track of the incoming `REQUEST` messages. This set is initialized to $\emptyset$, can grow when `REQUEST` messages are received, and is emptied when v gets matched.

We illustrate the basic communication involved in the parallel matching algorithm using the simple example shown in Fig. 3.1. In the figure, a complete graph on three vertices, u, v, w, with the weights of edges (u, v), (u, w) and (v, w) equal to 3, 2 and 1, respectively, is considered. Initially, $candidateMate(u) = v$, $candidateMate(v) = u$, $candidateMate(w) = u$, and $S(u) = \{v, w\}$, $S(v) = \{u, w\}$, $S(w) = \{u, v\}$.

The graph is depicted three times, corresponding to three different stages of the computation. During the first stage, all edges are drawn with dashed lines, to indicate that they are present and free. At this time, the initial messages $\texttt{REQUEST}(u, v)$, $\texttt{REQUEST}(v, u)$ and $\texttt{REQUEST}(w, u)$ are sent.

The former two `REQUEST` messages trigger the movement of edge (u, v) from E to M and the removal of edges (u, w) and (v, w) from E; this is indicated by drawing the edges with solid and dotted lines, respectively, in the last two stages of the computation. Every such handshake (two symmetric `REQUEST` messages) implies a locally dominant edge that must be added to M. Furthermore, $S(u)$ becomes $\{w\}$, $S(v)$ becomes $\{w\}$, and $\texttt{SUCCEEDED}(u, w)$ and $\texttt{SUCCEEDED}(v, w)$ messages are sent during the second stage of the computation, informing vertex w that vertices u and v are no longer available for matching and that w should look for some other candidate mate.

In this case w is left with no choice, $S(w)$ becomes empty and processor $p(w)$ stops listening for further messages since no edge incident on w is left. However, right before processor $p(w)$ ends its part of the computation, $\texttt{FAILED}(w, u)$ and $\texttt{FAILED}(w, v)$ messages are sent during the third stage of the computation, indicating that vertex w cannot be matched. This is required in order for $S(u)$ and $S(v)$ to be updated: they both become empty, which means that $p(u)$ and $p(v)$ stop listening as well.

Note that a slight variation of the computation is possible in this example, depending on the timing of the communication. In the scenario discussed so far there is an implicit

Algorithm 3.1 Parallel Matching

```
procedure ParallelMatching(v)
    mate(v) ← 0
    S(v) ← adj(v)
    R(v) ← ∅
    candidateMate(v) ← ComputeCandidateMate(v)
    if candidateMate(v) ≠ 0 then
        send REQUEST to candidateMate(v)
    while S(v) ≠ ∅ do
        receive message from u ∈ S(v)
        if message = REQUEST then
            ProcessRequestMessage(u, v)
        else if message = SUCCEEDED then
            ProcessSucceededMessage(u, v)
        else if message = FAILED then
            S(v) ← S(v) \ {u}
```

Algorithm 3.2 Process REQUEST Message

```
procedure ProcessRequestMessage(u, v)
    if mate(v) ≠ 0 then
        return
    if candidateMate(v) = u then
        mate(v) ← candidateMate(v)
        S(v) ← S(v) \ {mate(v)}
        R(v) ← ∅
        for w ∈ adj(v) \ {mate(v)} do
            send SUCCEEDED to w
    else
        R(v) ← R(v) ∪ {u}
```

assumption that message $\texttt{SUCCEEDED}(v, w)$ is received by processor $p(w)$ before message $\texttt{SUCCEEDED}(u, w)$, thus w is really left with no choice for a candidate mate. However, if the two SUCCEEDED messages arrive in reverse order, an opportunity is given to vertex w to consider vertex v as a candidate mate, resulting in an additional $\texttt{REQUEST}(w, v)$ message. The final outcome of the computation is, however, the same.

This simple parallel algorithm is formalized by procedure PARALLELMATCHING (Algorithm 3.1), which starts the computation for every vertex v and then keeps processing incoming messages as long as $S(v)$ is not empty. The processing is performed by two procedures PROCESSREQUESTMESSAGE and PROCESSSUCCEEDEDMESSAGE (Algorithms 3.2 and 3.3 respectively).

Procedure COMPUTECANDIDATEMATE, omitted here, simply computes $candidateMate(v)$ out of $S(v)$. Note that at least two and at most three messages are sent across any edge.

3.3. The General Parallel Algorithm

The general parallel algorithm (many vertices per processor), which is only briefly described here, is based on the procedures presented so far. We refer the reader to [9] for a detailed discussion of this algorithm and its analysis. In a similar work, Manne and Bisseling present a serial

Algorithm 3.3 Process SUCCEEDED Message

```
procedure ProcessSucceededMessage(u, v)
    S(v) ← S(v) \ {u}
    if mate(v) ≠ 0 then
        return
    if candidateMate(v) = u then
        candidateMate(v) ← ComputeCandidateMate(v)
        if candidateMate(v) ≠ 0 then
            send REQUEST to candidateMate(v)
            if candidateMate(v) ∈ R(v) then
                mate(v) ← candidateMate(v)
                S(v) ← S(v) \ {mate(v)}
                R(v) ← ∅
                for w ∈ adj(v) \ {mate(v)} do
                    send SUCCEEDED to w
        else
            for w ∈ adj(v) do
                send FAILED to w
```

implementation of the Hoepman's algorithm and discuss its parallelization using the Bulk Synchronous Parallel (BSP) model [18]. Our implementation is asynchronous and uses message-aggregation to optimize communication. In addition, we provide important implementation details.

We assume that the input graph is pre-distributed on a set of processors that will be involved in computing a matching. Cross edges are represented using *ghost* vertices: A boundary vertex u is stored on its corresponding processor $p(u)$ as well as on every other processor $p(v)$ such that (u, v) is a cross edge. On processor $p(v)$ vertex u represents a ghost vertex.

As before, for every vertex v the algorithm maintains the variable $candidateMate(v)$, which becomes the true mate of v, if v gets matched, or is 0 otherwise; it also maintains the set $S(v)$ that keeps track of all remaining edges incident on v. In this case, however, interior and cross edges are maintained separately.

The parallel computation interleaves local processing for the interior edges, with asynchronous point-to-point communication for the cross edges, handled with the same three types of messages. As before, at least two and at most three messages would be sent across any cross edge; however, in our algorithm, this number of messages is reduced substantially by bundling together messages sent between the same pair of processors, whenever possible. This is an important feature that distinguishes our algorithm from previous ones, and it makes it possible for the algorithm to scale to tens of thousands of processors.

The communication reduction is enabled by organizing the interleaved computation as a double loop, where the inner loop corresponds to the processing of interior vertices and the outer loop corresponds to the processing of the boundary vertices. The inner loop generates no messages in the algorithm, although an interior vertex might have to wait for a boundary vertex to be matched or become available to be matched, before it can decide on its mate.

The inner loop employs a queue Q of matched vertices to be processed by each processor, terminating when Q becomes empty. The outer loop seeks to match the boundary vertices, and might have to communicate with a processor owning a ghost vertex to find if a cross edge is available to be matched. In our implementation, the messages generated by all currently unmatched boundary vertices on a processor are sent in bundles to its neighboring processors. The termination of the outer loop is controlled by the number of cross edges remaining in the subgraph mapped to the corresponding processor. The number of iterations of the outer loop required for the parallel algorithm to terminate depends on the distribution of weights on the edges of the graph.

In a recent related work, Patwary *et al.* present a parallel greedy algorithm for the maximum (cardinality) matching problem [20]. We note that the maximum matching problem is different from the maximum edge-weighted matching problem.

4. Parallel Coloring

The parallel coloring algorithm for which results are presented in this paper is derived from the framework for parallel greedy coloring developed in [3]. We begin this section with a review of the essential features of the framework, and then point out aspects of the new, improved algorithm.

4.1. The Framework

By a *greedy* coloring algorithm is meant an algorithm that runs through the set of vertices in some *order* at each step assigning a vertex v the *smallest* color not used by any of the vertices adjacent to v (such a choice of color is referred to as *first-fit*). There exist several degree-based vertex ordering techniques that enable the greedy coloring algorithm to use near optimal number of colors [8]. The framework developed in [3] considered how the greedy coloring could be efficiently parallelized on distributed-memory architectures.

Given a graph partitioned among the processors of a distributed-memory machine, in the stated framework, each processor *speculatively* colors the vertices assigned to it in a series of rounds. Each round consists of a *tentative coloring* and a *conflict detection* phase. The coloring phase in a round is further broken down into *supersteps*. In each superstep, a processor first colors a pre-specified number $s \gg 1$ of its assigned vertices sequentially, using color information available at the beginning of the superstep, and only thereafter exchanges recent color information with other, relevant processors. The rationale for using infrequent, *coarse-grained* communication (rather than communication after a single vertex is colored) is to reduce the associated communication cost.

Algorithm 4.1 A Parallel Coloring Framework. *Input*: graph $G = (V, E)$ and superstep size s. *Initial data distribution*: V is partitioned into p subsets $V_1, \ldots, V_p$; processor P_i owns V_i, stores edges E_i incident on V_i, and stores the identity of processors hosting the other endpoints of E_i.

```
procedure ParallelColoring(G = (V, E), s)
  on each processor P_i, i ∈ I = {1, ..., p}
    for each boundary v ∈ V_i' = {u|(u, v) ∈ E_i} do
      Assign v a random number r(v) generated
        using v's ID as seed
    U_i ← V_i   ▷ U_i is the current set of vertices to be
      colored
    while ∃j ∈ I, U_j ≠ ∅ do
      if U_i ≠ ∅ then
        Partition U_i into ℓ_i subsets U_{i,1}, ..., U_{i,ℓ_i},
          each of size s
        for k ← 1 to ℓ_i do  ▷ each k corresponds
          to a superstep
          for each v ∈ U_{i,k} do
            assign v a "permissible" color c(v)
          Send colors of boundary vertices in
            U_{i,k} to other processors
          Receive color information from other
            processors
        Wait until all incoming messages are suc-
          cessfully received
        R_i ← ∅        ▷ R_i is a set of vertices to be
          re-colored
        for each boundary vertex v ∈ U_i do
          if ∃ (v, w) ∈ E_i where c(v) = c(w)
            and r(v) < r(w) then
            R_i ← R_i ∪ {v}
        U_i ← R_i
  end on
```

If two adjacent vertices owned by two different processors are colored in the same superstep, they may receive the same color and cause a *conflict*. In the conflict-detection phase, therefore, each processor examines those of its boundary 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. For a given conflict-edge—a cross edge whose endpoints have received the same color—it suffices to recolor either one of the endpoints to resolve the conflict. In order to achieve a balanced workload distribution across processors, the vertex to be re-colored is chosen *randomly*. For this purpose a random function is defined over boundary vertices at the beginning of the algorithm (this avoids the need for generating/communicating a random number each time it is needed). The conflict detection phase does not require

communication since by the end of the tentative coloring phase every processor has gathered complete information about the colors of the neighbors of its vertices. The scheme terminates when no more conflicts to be resolved remain. The framework is outlined more formally in Algorithm 4.1.

Several variant algorithms derived from this framework had been implemented using MPI and experimentally analyzed in [3]. The objective of the analysis was to determine the best way in which the various parameters of the framework need to be combined in order to reduce both runtime and number of colors. Among the questions the analysis attempted to answer are: How large should the superstep size s be? Should the supersteps be run synchronously or asynchronously? Should interior vertices be colored before, after, or interleaved with boundary vertices? How should a processor choose a color for a vertex (options of strategies here include first-fit, staggered first-fit, least-used etc)? Should inter-processor communication be customized or broadcast-based?

Experiments were carried out on two different platforms (PC clusters) using large-size synthetic as well as real graphs drawn from various application areas. The computational results obtained show that, for large-size well-partitioned graphs (have a small fraction of boundary vertices), a combination of parameters in which (i) a superstep size in the order of a thousand is used, (ii) supersteps are run asynchronously, (iii) each processor colors its assigned vertices in an order where interior vertices appear either strictly before or strictly after boundary vertices, (iv) a processor chooses a color for a vertex using a first-fit scheme (where the *smallest* available color is chosen at each coloring step), and (v) inter-processor communication is customized gives overall the best performance. This variant is called FIAC in [3].

For poorly-partitioned graphs (a majority of the vertices are boundary), good performance was observed to be attainable by using a superstep size close to a hundred in item (i), a broadcast based communication mode in item (v), and by keeping the remaining parameters as in the case with well-partitioned graphs. This variant is called FIAB in [3].

For most of the test graphs used in the experiments in [3], algorithms FIAC and FIAB converged rapidly—within at most six rounds—and yielded fairly good speedup when up to a few hundred processors are used. These algorithms were also compared with *maximal independent set*-based parallel coloring algorithms of the kinds suggested by Jones and Plassmann [11], and they were found to be consistently superior in performance. The main reason for the better performance is that the framework in Algorithm 4.1 uses provably fewer or at most as many *rounds* as the maximal independent set-based algorithm [3].

4.2. The New Algorithm

The algorithm presented in this paper is an enhancement of the FIAC scheme for scalability. Like FIAC, inter-processor communication in the new algorithm is customized, but it is performed in a different way. We explain next the difference between FIAC, the new algorithm and the broadcast based variant FIAB. Let B be the set of boundary vertices owned by a processor P_i, and let $B_S \subseteq B$ denote the vertices colored by P_i in a superstep S. We say a processor P_j is *neighbor* to the processor P_i if processor P_j owns at least one vertex that is adjacent to some vertex in B, and we call P_j *superstep-neighbor* to P_i if P_j owns at least one vertex that is adjacent to some vertex in B_S.

- In algorithm FIAB, processor P_i sends the same message, the union of the colors of the vertices in B_S, to *every other* processor P_k in the system.
- In algorithm FIAC, processor P_i again sends messages to *every other* processor P_k in the system, but each message (color information) is customized to the processor P_k. In the case where P_k does not own a vertex adjacent to some vertex in B_S, then the message would be "empty". Therefore, compared to FIAB, this reduces the total message volume, but not the number of messages.
- In the new algorithm, processor P_i sends customized messages to *every neighboring* processor P_j. Clearly, this reduces both the volume and the number of messages.

The implementation of the new algorithm is made available as part of the Zoltan parallel data management and load-balancing library [2].

5. Preliminary Experimental Results

We present in this section a small set of experimental results on the improved parallel distance-1 coloring algorithm just described and the parallel half-approximation algorithm for edge-weighted matching discussed in Section 3.

5.1. Experimental Setup

As inputs for this preliminary investigation on weak and strong scalability, we use a set of synthetically generated graphs and two real-world graphs derived from a circuit simulation application.

The synthetic graphs are five-point *grid graphs* that are model problems for partial differential equations. In these graphs, a node in a two-dimensional grid is connected to its neighbors to the east, west, north and south (except at the boundaries). These graphs are too simplistic to be reasonable choices for experiments on either coloring or matching in terms quality of solution (i.e. the number of colors used

Figure	Problem	Scaling	Input graph	Distribution	Max proc
Fig. 5.1	matching & coloring	Weak	$k \times k$ grid graphs, various k smallest: $\|V\| \approx 64M$, $\|E\| \approx 128M$ largest:$\|V\| \approx 1B$, $\|E\| \approx 2B$	Uniform 2D	16,384
Fig. 5.2	matching & coloring	Strong	$32,768 \times 32,768$ grid graph $32,000 \times 32,000$ grid graph $\|V\| \approx 1B$, $\|E\| \approx 2B$	Uniform 2D	16,384
Fig. 5.3	matching	Strong	Circuit simulation graph1 $\|V\| \approx 3.2M$, $\|E\| \approx 7.7M$	METIS (6% edge cut)	4,096
Fig. 5.4	coloring	Strong	Circuit simulation graph2 $\|V\| \approx 1.5M$, $\|E\| \approx 3M$ Max and Min degree: 6 and 2	ParMETIS (40% edge cut)	4,096

Table 5.1: Overview of experimental setup and results.

and the weight of the matching obtained). For example, exploiting the available structure, a five-point grid graph can be colored using just two colors. Similarly, for the matching case, a solution close to optimal could be obtained fairly easily, again exploiting the structure. We do not exploit the structure of the grid graphs in our experiments. Rather, we use the grid graphs to be able to conduct *weak scalability* study, where input size needs to be increased in proportion to the number of processors employed.

The grid graphs were generated *in parallel*, distributed in a two-dimensional fashion among the available processors. Each processor owns a subgraph corresponding to an appropriate portion of the grid and is responsible for computational tasks on the subgraph, in coordination with neighboring processors. For instance, in an experiment with an input grid graph of size $8,000 \times 8,000$ run on $1,024$ processors (a 32×32 processor-grid), each processor is assigned a subgraph corresponding to a 250×250 grid. As the input size is doubled, the number of processors is also doubled. Hence, ideally, the runtime is expected to remain constant. We considered $k \times k$ grid graphs for values of k ranging from $8,000$ to $32,000$. For the experiments on matching, the edges in the graphs were assigned *random* weights. This ensured that the grid structure did not play a significant role for the scalability study.

The two real-world test graphs, used in our experiments on strong scalability, are generated out of a matrix obtained from the University of Florida Sparse Matrix Collection. The matrix (named G_3 circuit) originates from a circuit simulation application. For the experiments on the matching algorithm, a *bipartite graph* representation of the matrix was used, whereas for the experiments on coloring an *adjacency graph* representation was used. For the experiments on matching, the graph was distributed among the processors using the serial partitioning software tool METIS [13]. For the experiments on the coloring algorithm, the graph was distributed among the processors using the parallel partitioning tool ParMETIS [14]. METIS generally gives much superior partitioning quality compared to ParMETIS on a large number of processors. We used these two different partitioning tools in our experiments, instead of just the one that is more favorable to our algorithms, to show that the algorithms deliver good performance even for relatively poorly distributed input data.

The coloring experiments were carried out using the Zoltan Toolkit [2] as a base framework. The experiments were performed on Intrepid, the IBM Blue Gene/P machine at Argonne's Advanced Leadership Computing Facility.

5.2. Results

The scalability results we obtained are summarized in Figure 5.1 through Figure 5.4. Table 5.1 gives a quick overview of the experimental setup along with some of the information contained in the figures. The compute times reported in Figures 5.1 to 5.4 concern total coloring or matching time, excluding the time taken to partition the input graph (in the case of the experiment on the circuit simulation graphs, Figures 5.3 and 5.4) or to do I/O.

On the grid graphs, it can be seen that both the matching and the coloring algorithm exhibited excellent scalability—both in terms of weak (Figure 5.1) and strong (Figure 5.2) scaling. The observed behavior is in part due to the near-ideal manner in which the grid graphs are distributed across processors.

The observed scalability in the experiment on the circuit simulation graphs (Figure 5.3 and Figure 5.4) is expectedly less than ideal, but still highly impressive, since the graph distribution across processors, attained here using the two graph partitioning tools METIS and ParMETIS, is far from ideal. For example, when partitioned with METIS on $4,096$ processors (as is done in the experiment on the matching algorithm), nearly 6% of the edges of the circuit simulation graph get cut (become cross edges). When partitioned using ParMETIS on the same number of processors (as is done in the experiment on the coloring algorithm), nearly 40% of the edges of the graph get cut. The relative performance degradation for large number of processors and the relative performance difference on the results on matching and coloring seen in Figures 5.3 and 5.4 reflect the impact cross edges have on overall performance.

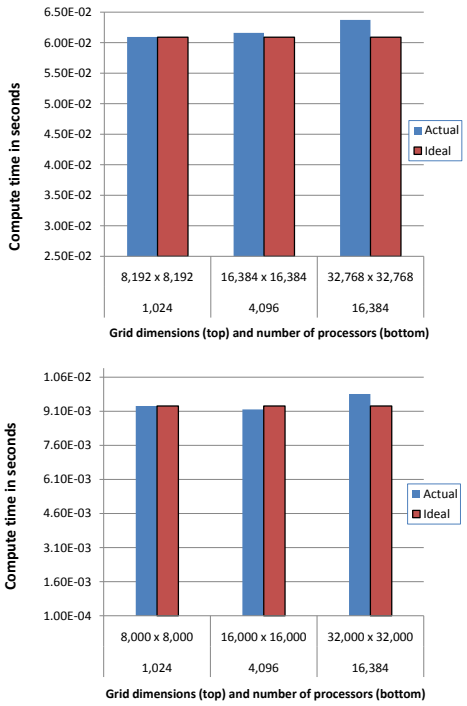

Figure 5.1: Weak scaling on various five-point grid graphs. Top, matching. Bottom, coloring.

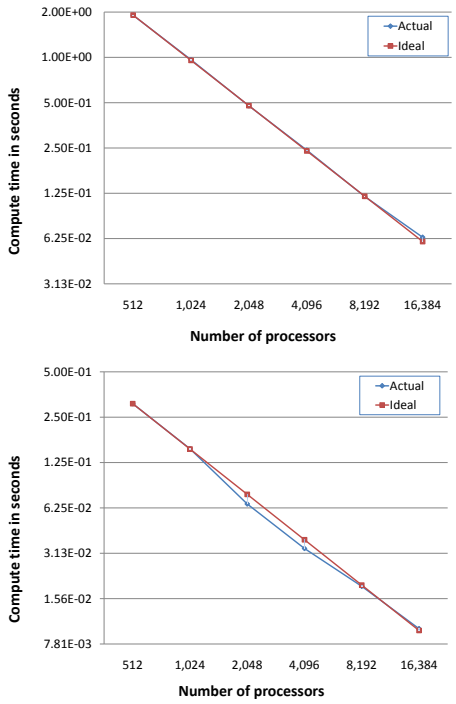

Figure 5.2: Top: Strong scaling of the matching algorithm on a five-point, $32,768 \times 32,768$ grid graph. Bottom: Strong scaling of the coloring algorithm on a five-point, $32,000 \times 32,000$ grid graph. Both axes are in log scale.

On both the grid graphs and the circuit simulation graph, the parallel matching algorithm yielded a solution in which the sum of the weights of edges in the computed matching remained the same, regardless of the number of processors used. The number of colors the parallel coloring algorithm uses varied slightly as the number of processors is varied, but in general remained nearly the same as the number used by the underlying serial algorithm.

6. Concluding Remarks and Outlook

We plan to present in future work more comprehensive experimental results and details that were omitted for space consideration.

Emerging many-core computing platforms are likely to be comprised of cores that in turn support parallelism via multithreading. Implementations that harness the full potential of such architectures will need to rely on the use of hybrid distributed-memory and shared-memory programming, for example, via the combined use of MPI and OpenMP. In future work, we will investigate how the coloring and matching algorithms developed here could be extended to employ such bi-level, hybrid programming approaches.

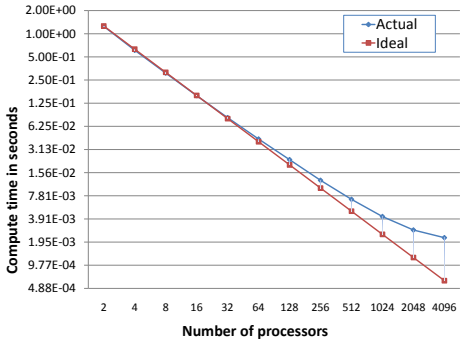

Figure 5.3: Strong scaling of the matching algorithm on a bipartite graph of a circuit simulation application (roughly 3.2 million vertices and 7.7 million edges; edge cut at 4096 processors: 6%.) Both axes are in log scale.

Acknowledgments. This research was supported by the U.S. Department of Energy through the CSCAPES Institute (grants DE-FC02-08ER25864 and DE-FC02-06ER2775), and by the National Science Foundation through grants CCF-0830645,CNS-0643969, OCI-0904809, OCI-0904802 and CNS-0403342. We are grateful to the National Energy Research Supercomputer Center (NERSC), the Advanced Leadership Computer Facility at Argonne National Lab, Ohio Supercomputer Center and the Rosen Center for Advanced Computing at Purdue University for access to the terascale computers on which we have run experiments.

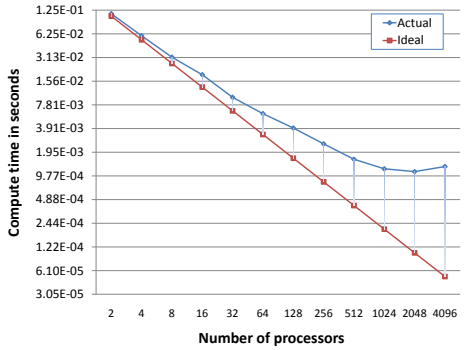

Figure 5.4: Strong scaling of the coloring algorithm on the adjacency graph of a circuit simulation application (roughly 1.5 million vertices and 3 million edges; edge cut at 4096 processors: 40%). Both axes are in log scale.

References

[1] B. Berger, R. Singh, and J. Xu. Graph algorithms for biological systems analysis. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 142–151, Philadelphia, PA, USA, 2008.

[2] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, V. Leung, L. A. Riesen, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, and J. Teresco. *Zoltan 3.0: Parallel Partitioning, Load Balancing, and Data-Management Services; User's Guide*. Sandia National Laboratories, Albuquerque, NM, 2007. Tech. Report SAND2007-4748W http://www.cs.sandia.gov/Zoltan/ug_html/ug.html.

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

[4] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, 1999.

[5] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.*, 22(4):973–996, 2000.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, 1979.

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

[8] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen. ColPack: Graph coloring software for derivative computation and beyond. Submitted to ACM Trans. on Math. Softw., 2010.

[9] M. Halappanavar. *Algorithms for vertex-weighted matching in graphs*. PhD thesis, Old Dominion University, Norfolk, VA, 2009.

[10] J.-H. Hoepman. Simple distributed weighted matchings. *CoRR*, cs.DC/0410047, 2004.

[11] M. Jones and P. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, 1993.

[12] M. Jones and P. Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Computing*, 20(5):753–773, 1994.

[13] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1), 1999.

[14] G. Karypis, K. Schloegel, and V. Kumar. ParMETIS: Parallel graph partitioning and sparse matrix ordering library, version 3.1. Technical report, Dept. Computer Science, University of Minnesota, 2003.

[15] V. S. A. Kumar, M. V. Marathe, S. Parthasarathy, and A. Srinivasan. End-to-end packet-scheduling in wireless ad-hoc networks. In *SODA 2004: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1021–1030, 2004.

[16] C. J. Langmead and B. R. Donald. High-throughput 3d structural homology detection via nmr resonance assignment. In *CSB '04: Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference*, pages 278–289, Washington, DC, USA, 2004. IEEE Computer Society.

[17] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[18] F. Manne and R. H. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *The Seventh International Conference on Parallel Processing and Applied Mathematics*, pages 708–717, 2007.

[19] A. Mehta, A. Saberi, U. Vazirani, and V. Vazirani. Adwords and generalized online matching. *J. ACM*, 54(5):22, 2007.

[20] M. M. A. Patwary, R. H. Bisseling, and F. Manne. Parallel greedy graph matching using an edge partitioning approach. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 45–54, New York, NY, USA, 2010. ACM.

[21] A. Pinar, E. Chow, and A. Pothen. Combinatorial algorithms for computing column space bases that have sparse inverses. *Electronic Transactions on Numerical Analysis*, 22:122–145, 2006.

[22] A. Pothen and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Softw.*, 16(4):303–324, 1990.

[23] R. Preis. Linear time $\frac{1}{2}$-approximation algorithm for maximum weighted matching in general graphs. In *16th Ann. Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 259–269, 1999.

[24] Y. Saad. ILUM: A multi-elimination ILU preconditioner for general sparse matrices. *SIAM J. Sci. Comput.*, 17:830–847, 1996.