10-1-1999

A Practical Parallel Algorithm for Cycle Detection in Partitioned Digraphs

D.A. Bader

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SCHOOL OF ENGINEERING UNIVERSITY OF NEW MEXICO

A Practical Parallel Algorithm for Cycle Detection in Partitioned Digraphs

David A. Bader[1]
dbader@eece.unm.edu
Department of Electrical and Computer Engineering
University of New Mexico, Albuquerque, NM 87131

UNM Technical Report: EECE-TR-99-004

Report Date: October 1999

[1]Supported in part by U.S. Department of Energy Sandia-University New Assistant Professorship Program (SUNAPP) Award # AX-3006.

Abstract

Graph theoretic techniques are used in variety of important computational problems in the areas of computational physics, mechanics, and fluid flow. We present a new, parallel algorithm for detecting cycles in partitioned, directed graphs that is both scalable in the graph and machine size, and performs well in practice. As an example, on a $p = 64$ processor cluster, we have solved an extremely large and difficult input problem with $n = 2^{28}$ vertices in less than five minutes. Our parallel algorithm uses a new graph representation, called Packed-Intervals, has a theoretical running time for this input of $\tau \log p + O\left(\frac{n}{p}\right)\sigma + O\left(\frac{n}{p}\right)$ for $n \geq p^4$, and achieves good speedup for any $n \gg p$ in practice. Our study includes both an efficient parallel algorithm and an experimental study.

Please see `http://www.eece.unm.edu/~dbader` for additional information.

Keywords

Cycle Detection, Graph Theory, Experimental Parallel Algorithms, Parallel Performance.

1 Introduction

Graph theoretic techniques are used in variety of important computational problems in the areas of computational physics, mechanics, and fluid flow. Efficient sequential algorithms for many of these graph-based techniques are well-known, for instance, searching, decomposing, and finding network flows. However, when either very large graphs require the memory space of parallel computers or faster solutions aim more processing power at the problem, efficient implementations of some of these same techniques for high-performance computers at this point are not known. One graph problem of significant interest is detecting cycles in directed graphs. This problem arises in the numerical stability of large-scale calculations that use finite-element meshes. Up to now, no efficient parallel implementations were known. In this paper, we present a new, parallel algorithm for detecting cycles in partitioned, directed graphs that is both scalable in the graph and machine size, and performs well in practice. Our study includes both an efficient parallel algorithm and an experimental study.

In a directed, planar graph (for example, a finite-element grid), with each arc directed from a source to destination vertex, **detecting cycles** takes linear sequential time in the size of the graph. On a parallel machine, we will assume that a tool such as MeTiS [9] has partitioned the input graph *a priori* to the processors in a manner that maximizes the locality of each processor's subgraph.

Our novel approach contains three phases. During the first phase, each node's subgraph is locally examined for cycles. Arcs spanning node boundaries are discovered in the second phase. Finally, in the third phase, pairwise merging between subgraphs reduces the problem to a single processing node, all-the-while examining the graph for cycles and aggressively pruning vertices and incident arcs that do not contribute to cycles. The merging phase may be prohibitively expensive using a standard graph representation (herein referred to as the "Express" approach) that reduces each subgraph to boundary vertices, subgraph-arcs directly connecting the boundaries, and arcs spanning node boundaries. We have discovered a novel approach that uses what we call the compact "Packed-Intervals" graph representation. Instead of recording each subgraph-arc, our algorithm records only the endpoint labels for each interval of reachable boundary vertices. Our parallel implementation using the Packed-Intervals graph representation now outperforms the sequential approach with nearly linear speedup in the number of processors and problem size (for example, see Figure 4) and has a complexity cost of $\tau \log p + \mathrm{O}\left(\frac{n}{p}\right)\sigma + \mathrm{O}\left(\frac{n}{p}\right)$ for instances of large meshes typically seen in practice. Our parallel implementation proves the capability of solving extremely large inputs that, to our knowledge, have never been attempted before. For instance, on a 64-processor cluster, our implementation solves an input graph with 16M ($\mathrm{M} \equiv 2^{20}$) vertices in 12.2 seconds, with 64M vertices in 48.0 seconds, and with 256M vertices in 288 seconds. Thus, our new algorithm solves problem instances once thought impossible and scales almost linearly with problem size on a large cluster.

The main contributions of this paper are

- an efficient parallel algorithm for detecting cycles in well-partitioned planar graphs,
- a new graph representation for well-partitioned planar graphs, and
- an experimental parallel algorithmic framework for performance-engineering of graph theoretic problems.

The organization of our paper is as follows. Section 2 describes our realistic model for parallel computation. Both sequential and parallel cycle detection algorithms are presented in Section 3. Section 4 contains the theoretical and practical analyses for these algorithms. Finally, Section 5 discusses our results and future directions.

2 The Parallel Computation Model

We use a simple model to analyze the performance of our parallel algorithms. Our model is based on the fact that current hardware platforms can be viewed as a collection of powerful processors connected by a communication

network that can be modeled as a complete graph on which communication is subject to the restrictions imposed by the latency and the bandwidth properties of the network. We view a parallel algorithm as a sequence of local computations interleaved with communication steps, where we allow computation and communication to overlap. We account for communication costs as follows.

Assuming no congestion, the transfer of a block consisting of m contiguous words between two processors takes $(\tau + \sigma m)$ time, where τ is the latency of the network and σ is the time per word at which a processor can inject or receive data from the network. Note that the bandwidth per processor is inversely proportional to σ. We assume that the bisection bandwidth is sufficiently high to support block permutation routing amongst the p processors at the rate of $\frac{1}{\sigma}$. In particular, for any subset of q processors, a block permutation amongst the q processors takes $(\tau + \sigma m)$ time, where m is the size of the largest block. Such cost (which is an overestimate) can be justified by using our earlier work [2, 8].

Using this cost model, we can evaluate the communication time $T_{comm}(n,p)$ of an algorithm as a function of the input size n, the number of processors p, and the parameters τ and σ. The coefficient of τ gives the total number of times collective communication primitives are used, and the coefficient of σ gives the maximum total amount of data exchanged between a processor and the remaining processors.

This communication model is close to a number of similar models (e.g. [5, 13, 1]) that have recently appeared in the literature and seems to be well-suited for designing parallel algorithms on current high performance platforms.

We define the computation time $T_{comp}(n,p)$ as the maximum time it takes a processor to perform all the local computation steps. In general, the overall performance $T_{comp}(n,p) + T_{comm}(n,p)$ involves a tradeoff between $T_{comm}(n,p)$ and $T_{comp}(n,p)$. Our aim is to develop parallel algorithms that achieve $T_{comp}(n,p) = \mathrm{O}\left(\frac{T_{seq}}{p}\right)$ such that $T_{comm}(n,p)$ is minimum, where T_{seq} is the complexity of the best sequential algorithm. Such optimization has worked very well for the problems we have looked at, but other optimization criteria are possible. The important point to notice is that, in addition to scalability, our optimization criterion requires that the parallel algorithm be an efficient sequential algorithm (i.e., the total number of operations of the parallel algorithm is of the same order as T_{seq}).

3 Cycle Detection

Given a *directed graph* (or *digraph*) $G = (V,E)$, the cycle detection algorithm determines whether a *simple cycle* can be found in this graph. An *arc* $e = (v_i, v_t) \in E$ is a directed edge that connects the *initial vertex* v_i at the tail to the *terminal vertex* v_t at the head ($v_i, v_t \in V$). A *directed path* (or *dipath*) in G is an ordered set of vertices $\{v_0, v_1, v_2, \ldots, v_{l-1}\}$ such that $\forall i \in [0, l-2]$, $(v_i, v_{i+1}) \in E$. A *simple dipath* does not re-visit any of the vertices. A simple cycle is a dipath with an additional arc $(v_{l-1}, v_0) \in E$ that returns to the starting vertex v_0. Let the number of vertices $n = |V|$ and the number of arcs $m = |E|$.

On a sequential machine, cycles can be detected in an arbitrary digraph by applying a well-known search technique as follows. An arbitrary vertex is selected and used as the starting point for a depth-first search. If no cycles are detected, but unvisited vertices still exist in the graph, the process is repeated from any one of the unvisited vertices. This search technique traverses the entire graph in $\mathrm{O}(n+m)$ time, classifying edges as *tree*, *forward*, *cross*, and *back*, whereby a tree edge is the first discovered edge leaving a vertex, a forward edge connects a vertex to an already discovered descendent vertex, a cross edge connects siblings in a tree or two trees together, and a back edge connects a vertex to its ancestor. A digraph is acyclic if and only if no back edges are discovered. [4]

3.1 Parallel Algorithm

On a parallel machine, we will assume the input graph has been partitioned *a priori* and evenly to the processors. The partition function f maps each vertex $v \in V$ to a processor p_z, for some $(0 \leq z \leq p-1)$, namely $f(v) = p_z$, and minimizes the number of arcs spanning across processor boundaries. Thus, the partition also maximizes the locality of each processor's subgraph. For example, MeTiS [9] is a software package that uses multi-level graph techniques to quickly partition an unstructured graph consistent with this criteria.

One simple method for parallelizing this detection algorithm could use the depth-first search technique as before, except when a terminal vertex of an arc in the search path is assigned to a different processor, the appropriate inter-processor messages are sent to continue the search on the new processor. However, this would perform very poorly for two reasons. First, in the worst case, a search path could traverse processor boundaries on every step of the path, generating $O(n)$ messages in sequential succession. Second, as the search progresses, only one processor at a time is busy working, while the others sit idle. Thus, there would be no parallel speedup. In fact, no efficient parallel implementations are known at this point for depth-first graph traversal methods [7].

3.2 A Possible Approach

Our novel approach attempts to minimize the communication while improving the load balancing by keeping as many processors as possible busy with useful work. There are three major phases in the algorithm. During the first phase, each processor examines its local subgraph and detects local cycles. The second phase constructs a reduced graph by aggressively collapsing and pruning the input graph. And the third stage merges this distributed graph until a cycle is detected or no cycles are found. Without loss of generality, we will assume that the indegree and outdegree of each vertex is bounded by a small constant (say, 6).

We present two new parallel algorithms that are similar, but use different graph representations for the second and third phases. Our first approach in Section 3.3 uses a graph representation that we call the ***Express*** approach. We describe the Express algorithm first because of its straightforward design, but note that this approach lacks parallel speedup. In Section 3.4, we present a major improvement using a novel graph representation that we call the ***Packed-Intervals*** approach. The Packed-Intervals algorithm uses a novel graph representation that significantly improves the algorithmic cost and scales efficiently with machine and problem size.

Next we define several terms that will be used and further explained in our algorithm.

Definitions

As mentioned earlier, the input digraph is spread amongst the processors using a partition of the vertices. Let $f(v)$ be the function that maps each vertex $v \in V$ to a processor, and let V_z be the set of vertices mapped to processor p_z. Each arc $(v_i, v_t) \in E$ is assigned to the processor that holds its initial vertex, $f(v_i)$, and let E_z be the set of all arcs incident from vertices assigned to processor p_z. In this manner, two types of arcs can be identified. A *local-arc* has both the initial and terminal vertices assigned to the same processor $(f(v_i) \equiv f(v_t))$, while a *trans-arc* spans across two different processors, $(f(v_i) \neq f(v_t))$. An *initial trans-arc* w.r.t. processor p_z has $f(v_i) \equiv p_z$, and a *terminal trans-arc* w.r.t. processor p_z has $f(v_t) \equiv p_z$. Note that the input vertices are distributed evenly across the processors, and only information about the local vertices, and arcs incident from these vertices (including the processor id of each initial trans-arc's terminal vertex), are held on each processor.

The second phase of this algorithm builds a new digraph, called an *express graph*. Each processor p_z will create two types of vertices, an *entrance vertex* (one for each terminal trans-arc) and an *exit vertex* (one for each initial trans-arc), w.r.t. processor p_z. There will also be two types of arcs in the express graph, trans-arcs from the original digraph, and *express arcs* with initial and terminal vertices corresponding to entrance and exit vertices, respectively. The Express algorithm in Section 3.3 uses a graph representation that explicitly contains these express arcs, while the Packed-Interval approach in Section 3.4 uses a new graph representation that implicitly holds this information in a compact data structure.

We are now ready to describe the three major phases of the parallel cycle detection algorithms.

- **Phase (1)**: (Discovery) Because of the distributed graph layout, each processor needs to find the set of trans-arcs that have a terminal vertex assigned to it. During this discovery process, the algorithm aggressively prunes local-arcs and vertices that are determined to be acyclic. If a cycle is found embedded in the subgraph induced by the set of local-arcs, the algorithm halts.
- **Phase (2)**: (Express) The express graph is constructed in parallel. For each initial (terminal) vertex of each trans-arc, an exit (entrance) vertex is created in the express graph and assigned to the same processor. Each trans-arc from the original graph is transferred to the corresponding exit and entrance vertices of the express graph. For each entrance vertex, an express arc is added to the graph (using the appropriate representation, as we will discuss in the following sections), whenever a dipath exists between the corresponding vertices in the original local subgraph.

  Notice that each express subgraph induced by a processor's express arcs is a *directed bipartite graph*, with each arc directed from the initial set of entrance vertices to the terminal set of exit vertices.
- **Phase (3)**: (Merge) The goal of this merging step is to iteratively combine pairs of express subgraphs until only a single graph remains on one processor. During the merging of these subgraphs, vertices and arcs that do not contribute to cycles are aggressively pruned, and the merged subgraphs are tested for the presence of cycles. The optimal merging scheme would pair together express subgraphs that contain the most connectedness between them (that is, the greatest number of trans-arcs incident to both subgraphs). We could examine all remaining express subgraphs, count the number of trans-arcs spanning between each pair, and then find the maximum weight perfect matching of these graphs. For κ express graphs, the running time is $O(\kappa^3)$ [10, 11]. Greedy approaches (on the subset of graphs that obey the triangle-inequality) that repeatedly match the two best unmatched points have $O(\kappa^2 \log \kappa)$ running time, and are not guaranteed to perform well [12]. While this matching can be implemented in polynomial time, our solution uses a simple heuristic that pairs express subgraphs by proximity to each other, with the partitioning assumption that most trans-arcs span neighboring express subgraphs. We expect this to perform well in practice and simply the task of tracking vertex-to-processor assignments during the merge phase.

```
1 merge(G_Z)
2     for h ⟵ 0 to log p − 1 do
3         if last(z , h) ≡ 0 then
4             if test(z , h) ≡ 0 then
5                 Recv(G_set(z,h)) from processor p_set(z,h) ;
6                 G_Z ⟵ MergeGraph( G_Z, G_set(z,h), h ) ;
              else
7                 Send(G_Z) to processor p_clear(z,h) ;
```

Algorithm 1: Merge phase algorithm for processor p_Z. Note that the **`MergeGraph`** routine calls the appropriate subgraph merging procedure for either Express or Packed-Interval graphs.

Algorithm (1) performs the iterative merging of the p express subgraphs. In this algorithm, we use four bit-oriented functions: `last`(α, b) which returns the b least-significant bits of α, `test`(α, b) which returns the b^{th} least-significant bit of α, `set`(α, b) which returns α with the b^{th} least-significant bit of α set to 1, and `clear`(α, b) which returns α with the b^{th} least-significant bit of α cleared to 0.

3.3 Express Graph Representation

The Express algorithm uses a straightforward data structure for representing the express graph in the second and third phases. This data structure explicitly holds the initial and terminal vertices, express arcs, and trans-arcs.

```
1  discovery(G_Z)
2      Q_Z ⟵ ∅ ;
3      foreach local vertex v ∈ V_Z do
4          color[v] ⟵ white ;

5      foreach local vertex v ∈ V_Z do
6          if color[v] ≡ white then visit(v) ;

7      sort Q_Z with the terminal vertex's processor number as the sorting key ;
8      communicate: exchange arc information with any processor having trans-arcs incident from or to p_Z;

9  visit(v)
10     R_v ⟵ ∅ ;
11     color[v] ⟵ red ;
12     foreach local-arc incident from v to w do
13         switch color[w] do
14             case white
15                 R_w ⟵ visit(w) ;
16                 R_v ⟵ R_v ∪ R_w ;
17             case black
18                 R_v ⟵ R_v ∪ R_w ;

19             case red
20                 HALT: A cycle has been detected ;

21     foreach trans-arc incident from v to w do
22         Q_v ⟵ Q_v ∪ (v, w) ;
23         R_v ⟵ R_v ∪ {v} ;
24     if R_v ≡ ∅ then
25         color[v] ⟵ green ;
       else
26         color[v] ⟵ black ;
27     return R_v
```

Algorithm 2: Discovery phase algorithm for Express graphs run independently on each processor p_Z.

Algorithm (2) performs the discovery phase on each processor p_Z, $(0 \leq z \leq p-1)$, and exchanges trans-arc information with each processor that shares an incoming or outgoing trans-arc. Prior to the depth-first search, local vertices are initially colored white. The first time a vertex $v \in V_Z$ is visited, its color is updated to red. After visiting all of vertex v's children, if a trans-arc is incident from v or any of its children, v is colored green, otherwise v is colored black.

During the second phase, the express graph (ExG) is constructed in parallel. For each initial (terminal) vertex of each trans-arc, an exit (entrance) vertex is created in the express graph and assigned to the same processor. Each trans-arc from the original graph is transferred to the corresponding exit and entrance vertices of the express graph. For each entrance vertex, an express arc is explicitly added to each local exit vertex, whenever a dipath exists between the corresponding vertices in the original local subgraph. This reachability information is contained in the set R_v for each black-colored vertex $v \in V$, collected during the discovery phase. All green-colored vertices and incident arcs can effectively be removed at this time, since each dipath starting from a green vertex has been explored and determined to be acyclic in the discovery phase. Each exit vertex in the express graph will also hold a list of its predecessor entrance vertices. This predecessor list will be used in the merging phase.

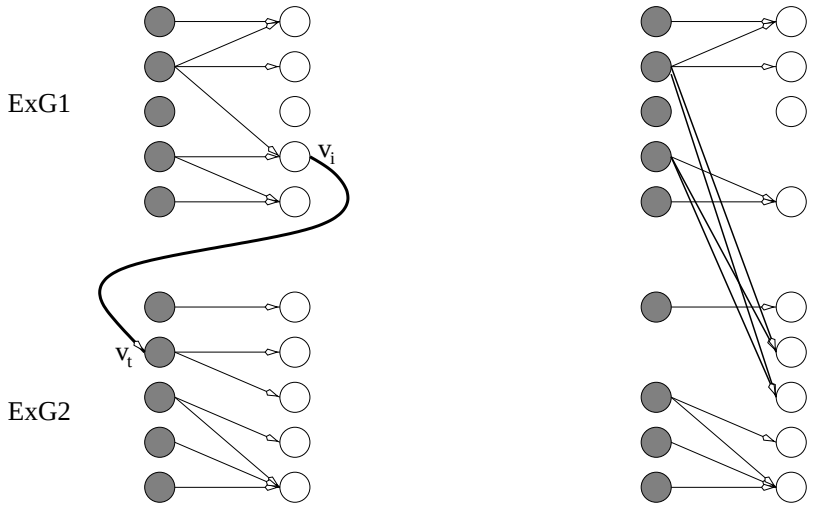

Figure 1: On the left, we show two express subgraphs, $ExG1$ and $ExG2$, before the merge phase. Note that we have selected one trans-arc in particular that connects exit vertex v_i from $ExG0$ to entrance vertex v_t in $ExG1$, but other trans-arcs may exist. On the right, we show the merged express graph, $ExG0$. The shaded (white) vertices are entrance (exit) vertices.

The heart of the merge routine in the third phase, described in Algorithm (3), reduces two express subgraphs in the Express graph representation into a single express subgraph. The merging checks the union of the two subgraphs for each trans-arc that has both initial and terminal vertices co-located in the union. If and when such

```
1  MergeGraph-Express(ExG1, ExG2, step)
   { Merge together express subgraphs ExG1 and ExG2 } ;

       { The vertex set of the result is the union of the input graphs' vertices } ;
2      V[ExG0] ⟵ V[ExG1] ∪ V[ExG2] ;
3      transfer each express arc from ExG1 and ExG2 to the corresponding vertices in ExG0 ;
4      foreach initial trans-arc in ExG1 and ExG2 do
           { The terminal vertex v_t, originally assigned to processor p_α, after step merging steps, (0 ≤ step ≤
           log p − 1), is now located on processor p_⌊α/2^(step+1)⌋ } ;
5          if the terminal vertex v_t ∉ V[ExG0] then
6              transfer this trans-arc to the corresponding exit vertex in ExG0 ;
           else
               { assume w.l.o.g. the trans-arc (v_i, v_t) exits ExG1 and enters ExG2 } ;
7              if (v_t, v_i) is an express arc in ExG0 then
8                  HALT: A cycle has been detected ;
               { Note that v_t may have no predecessors or successors } ;
9              foreach predecessor (v_p) of v_i do
10                 foreach successor (v_s) of v_t do
11                     add a new express arc (v_p, v_s) to ExG0 ;
12             remove vertices v_i and v_t and all incident edges from ExG0 ;
13     return ExG0 ;
```

Algorithm 3: Express subgraph merge algorithm.

trans-arcs are found, the terminal vertices that precede the initial vertex of the trans-arc are connected by express arcs to the successors of the terminal vertex of the trans-arc. Next, the trans-arc, its corresponding vertices, and their incident express arcs are all removed from the express graph. Figure 1 shows an example of one step of this merging. During each merge step, cycles from the original graph are detected whenever one of the trans-arcs found above has a terminal vertex with an express arc to its initial vertex. If a cycle is found, the algorithm terminates and reports the trans-arc that is, in fact, an arc from the original graph.

3.4 Packed-Intervals Graph Representation

The previous approach using the express graph representation during the merging stage does not scale as the problem size increases. In the first merging step with $O\left(\sqrt{\frac{n}{p}}\right)$ entrance and exit vertices, it is quite possible to have each entrance vertex reach many of the exit vertices. This results in $O\left(\frac{n}{p}\right)$ express arcs, which is asymptotically equivalent to the size of the original graph. We can significantly improve this by noticing that in a partitioned planar digraph, the resulting local subgraph is both bipartite and convex. That is, the entrance and exit vertex sets partition the vertices into two sets such that each express arc in the subgraph has a tail in the entrance vertex set and a head in the exit vertex set. A *convex bipartite graph*, as defined by Glover [6], holds the property that the set of exit vertices that are adjacent to each entrance vertex are labeled consecutively. Thus, the convex bipartite graph can be represented by an interval $[v_\alpha, v_\beta]$ for each entrance vertex v_e, meaning that vertex v_e has an express arc to each of the consecutively labeled exit vertices $[v_\alpha, v_{\alpha+1}, v_{\alpha+2}, \cdots, v_\beta]$.

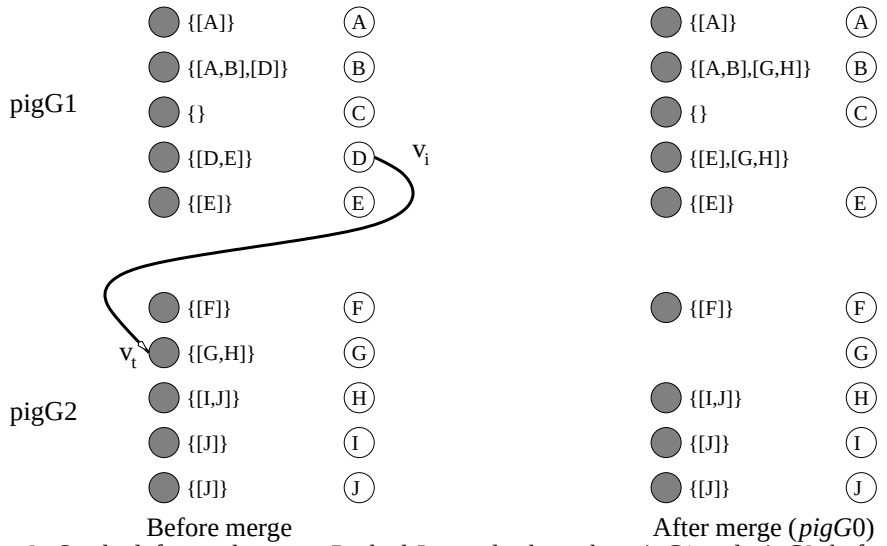

Figure 2: On the left, we show two Packed-Interval subgraphs, *pigG*1 and *pigG*2, before the merge phase. Note that we have selected one trans-arc in particular that connects exit vertex v_i from *pigG*0 to entrance vertex v_t in *pigG*1, but other trans-arcs may exist. On the right, we show the merged express graph, *pigG*0. The shaded (white) vertices are entrance (exit) vertices.

After each merging step, the resulting subgraph again is bipartite, but now each entrance vertex may have a list of intervals. In fact, each entrance vertex after merging step i, $(1 \leq i \leq \log p)$, could hold up to 2^i intervals, hence the name Packed-Intervals. Figure 2 shows an example of a merging step using the Packed-Interval graph representation. Unlike the Express graph approach, express arcs are not explicit objects in the data structure. Instead, each entrance vertex holds a compact list of exit vertex interval pairs.

If there are gaps in the labeling of the original intervals, then the exit vertices can be simply relabeled to use the compact convex graph representation. In a planar digraph, this relabeling will always be possible. The proof by contradiction is as follows. Suppose there is no way to relabel the exit vertices to satisfy the convex graph

```
1  visit(v)
2      v_α ⟵ ∅ ; v_β ⟵ ∅ ;
3      color[v] ⟵ red ;
4      foreach local-arc incident from v to w do
5          switch color[w] do
6              case white
7                  (w_α, w_β) ⟵ visit(w) ;
8                  v_α ⟵ min(v_α, w_α) ; v_β ⟵ max(v_β, w_β) ;
9              case black
10                 v_α ⟵ min(v_α, w_α) ; v_β ⟵ max(v_β, w_β) ;
11             case red
12                 HALT: A cycle has been detected ;
13     foreach trans-arc incident from v to w do
14         Q_v ⟵ Q_v ∪ (v, w) ;
15         if v < v_α or v_α ≡ ∅ then
16             v_α ⟵ v ;
17         if v > v_β or v_β ≡ ∅ then
18             v_β ⟵ v ;
19     color[v] ⟵ black ;
20     return (v_α, v_β)
```

Algorithm 4: Discovery phase algorithm for Packed-Interval graphs run independently on each processor p_Z. For simplicity, assume that the $\min(x,y)$ and $\max(x,y)$ functions become an assignment to y when $x \equiv \emptyset$.

representation. Then, we have entrance vertex v_0 that reaches exit vertices v_α through v_β, except for at least one exit vertex v_x with label between v_α and v_β. v_x must be reachable by at least one other entrance vertex, say v_1, or else v_x could have been trivially relabeled outside of this range. By planarity, v_0 has a path to v_α and a path to v_β, but no path to v_x. v_1 has a path to v_x that does not cross either of the two paths from v_0. This is a contradiction, though, since the path from v_1 to v_x must cross one of the two paths from v_0. Thus, the subgraph is representable using convex notation. In fact, we will assume that the graph partitioning performed *a priori* to the cycle detection provides this labeling, as it is commonly used in the partitioning heuristics.

4 Parallel Cycle Detection Complexity Analysis

In Section 4.1 we first describe the class of input graphs that will be used to analyze both the theoretical and empirical performance of our cycle detection algorithms. Section 4.2 and 4.3 present the theoretical analyses of the parallel algorithms for the Express and Packed-Interval graph representations, respectively. Finally, Section 4.4 reports on the experimental results for both approaches.

4.1 Experimental Input Graph

The analysis and running times for the parallel cycle detection algorithms are strongly dependent on the input. To illicit a worst-case analysis and performance for a typical and difficult class of input graphs, we use the following family of graphs. Let p be the number of processors and $n = 2^{2d}$ be the number of vertices in the directed, planar graph, for $d \geq \log_2 \sqrt{p}$. The vertices are arranged in a regular, two-dimensional square mesh with $\sqrt{n}$ vertices

```
1  MergeGraph-PIG(PIG_G1, PIG_G2, step)
   { Merge together Packed-Interval subgraphs PIG_G1 and PIG_G2 } ;

       { The vertex set of the result is the union of the input graphs' vertices } ;
2      V[PIG_G0] ⟵ V[PIG_G1] ∪ V[PIG_G2] ;
3      transfer the Packed-Intervals from PIG_G1 and PIG_G2 to the corresponding vertices in PIG_G0 ;
4      foreach initial trans-arc in PIG_G1 and PIG_G2 do
           { The terminal vertex v_t, originally assigned to processor p_α, after step merging steps, (0 ≤ step ≤
           log p − 1), is now located on processor p_⌊α/2^(step+1)⌋ } ;
5          if the terminal vertex v_t ∉ V[PIG_G0] then
6              transfer this trans-arc to the corresponding exit vertex in PIG_G0 ;
           else
               { assume w.l.o.g. the trans-arc (v_i, v_t) exits PIG_G1 and enters PIG_G2 } ;
7              if i is contained in any of v_t's intervals then
8                  HALT: A cycle has been detected ;
               { Note that v_t may have no predecessors or Packed-Intervals } ;
9              foreach predecessor (v_p) of v_i do
10                 combine the intervals from v_i into those of v_p;
11             remove vertices v_i and v_t. ;
12             if a Packed-Interval in PIG_G0 is lower- (upper-) bounded by α (β) then
13                 the interval endpoint should be incremented (decremented) ;
14             remove any interval where endpoint α > β ;
15     return PIG_G0 ;
```

Algorithm 5: Packed-Interval subgraph merge algorithm.

on each side. The blocks of $\sqrt{\frac{n}{p}} \times \sqrt{\frac{n}{p}}$ vertices are partitioned one to a processor, using row-major assignment of the blocks to processors. Each vertex has arcs to the vertices to its east and south, if the vertices exists. An additional arc is added from the southeast-most vertex to the northwest-most vertices, creating cycles of length at least $\sqrt{2n}$. The vertices in block j on processor j, $(0 \le j \le p-1)$, are labeled from $j\frac{n}{p}$ to $(j+1)\frac{n}{p}-1$, by

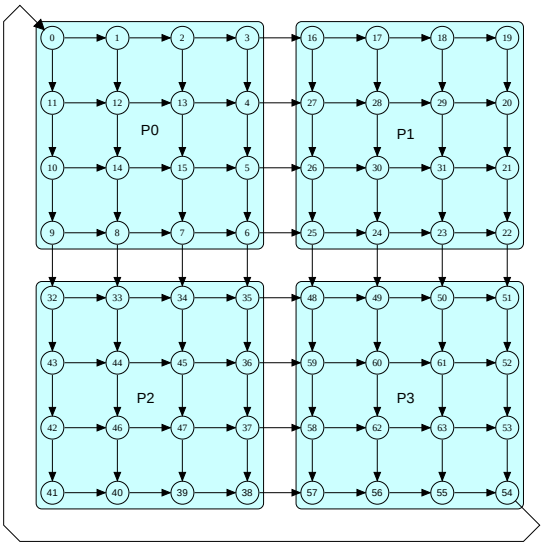

Figure 3: Example of a Lattice Input Graph with $n = 64$ and $p = 4$.

first consecutively labeling the border vertices, then the internal vertices. Figure 3 shows an example of this input graph for $n = 64$ vertices and $p = 4$ processors. Both the partitioning and the labeling are determined *a priori* by the graph partitioning tool such as MeTiS.

4.2 Express Graph Representation Analysis

The running time of the Express algorithm is strongly dependent on the input graph, and we consider the family of typical graphs described in the previous section. In this case, each processor holds a sub-lattice of $\frac{n}{p}$ vertices with $O\left(\sqrt{\frac{n}{p}}\right)$ boundary vertices.

Each processor independently runs the discovery phase with time complexity as follow. The initial coloring of vertices takes $O\left(\frac{n}{p}\right)$ time. The visiting of each vertex takes $O\left(\frac{n^2}{p^2}\right)$ to construct the reachability sets. This worst case time is tight when we consider an input of a chain of vertices, each with a constant number of initial trans-arcs $k_i \leq 6$. In this case, the reachability sets will have total size of $\sum_{i=1}^{\frac{n}{p}} \sum_{r=i}^{\frac{n}{p}} k_r \leq 6 \sum_{i=1}^{\frac{n}{p}} i = O\left(\frac{n^2}{p^2}\right)$. Sorting the initial trans-arcs (at most a constant number k per vertex) by terminal vertex processor number takes $O\left(\frac{n}{p}\right)$ using radix sort. Finally, exchanging trans-arc information takes $\tau + \frac{n}{p}\sigma$ communication cost. Thus, the total time for the discovery phase is $\tau + O\left(\frac{n}{p}\right)\sigma + O\left(\frac{n^2}{p^2}\right)$.

The express graph construction in the second phase takes $O\left(\frac{n^2}{p^2}\right)$ time, since there are at most $\frac{n}{p}$ entrance and $\frac{n}{p}$ exit vertices, and no more express arcs than the product of these two.

To bound the cost of the third phase, we assume that in the worst case, during each merging step, subgraphs that have no trans-arcs in common may be sent between processors. Thus, during each successive merge step, the size of each subgraph would double. There are a total of $\log p$ merging steps, with step i, $(1 \leq i \leq \log p)$, taking $\tau + S_{exp}(i)\sigma$ communication and $O(S_{exp}(i))$ computation time, where $S_{exp}(i)$ is the size of the largest subgraph during step i. During step i, there are $O\left(2^{\frac{i}{2}}\sqrt{\frac{n}{p}}\right)$ entrance and exit vertices, and $O\left(2^i \frac{n}{p}\right)$ express arcs. Therefore, the total merging complexity is $\tau \log p + O\left(\frac{n}{\sqrt{p}}\right)\sigma + O\left(\frac{n}{\sqrt{p}}\right)$.

Thus, the total complexity for cycle detection using the express graph representation is $\tau \log p + O\left(\frac{n}{\sqrt{p}}\right)\sigma + O\left(\frac{n^2}{p^2}\right)$ for $n \geq p^{\frac{3}{2}}$.

4.3 Packed-Interval Graph Complexity Analysis

The modified `visit` procedure for Packed-Interval graphs is described in Algorithm (4). Algorithm (5) presents the new method for merging two Packed-Interval subgraphs.

The Packed-Intervals algorithm costs $O\left(\frac{n}{p}\right) + \tau + O\left(\frac{n}{p}\right)\sigma$ in the discovery phase, and $O\left(\frac{n}{p}\right)$ to construct the Packed-Intervals graph in the second phase. The merging during step i, $(1 \leq i \leq \log p)$, assuming the worst case as in the analysis of the express graph approach, has communication cost $\tau + S_{pig}(i)\sigma$ and, because the merging cost of two Packed-Interval subgraphs is linear in the size of the two subgraphs, $O(S_{pig}(i))$ computation, where $S_{pig}(i)$ is the size of the largest subgraph during step i. During step i, there are $O\left(2^{\frac{i}{2}}\sqrt{\frac{n}{p}}\right)$ entrance and exit vertices, and in a significant reduction from the express graph approach, the arc information can be encoded in Packed-Intervals using $O\left(\left(2^{\frac{i}{2}}\sqrt{\frac{n}{p}}\right)2^i\right)$ space. Thus, the total merging cost is $\tau \log p + O(p\sqrt{n})\sigma + O(p\sqrt{n})$.

Therefore, the total complexity for cycle detection using our novel Packed-Intervals graph representation is $\tau \log p + O\left(\frac{n}{p}\right)\sigma + O\left(\frac{n}{p}\right)$ for $n \geq p^4$. As we will demonstrate experimentally in the next section, because this is

a worst-case derived bound, any $n \gg p$ achieves good speedup in practice.

4.4 Experimental Results

In this section, we compare and evaluate the empirical performance of our Express and Packed-Interval graph representation algorithms for cycle detection with the theoretical analyses. Our implementations use the standard message passing interface (MPI), and ran on the National Computational Science Alliance / University of New Mexico "Roadrunner" Linux SuperCluster [3]. Roadrunner contains 64 SMP nodes (128 processors) interconnected by Myrinet, a high-performance gigabit switched network. Each node in the cluster contains dual 450 MHz Intel Pentium II processors with 512MB RAM and 6.4GB hard disk. Our MPI implementation uses the Portland Group C compiler, MPICH-GM version 1.1.2.4 (the Myricom extension of MPICH for Myrinet), Myrinet GM drivers 1.0.4, Redhat Linux 5.2, and the Linux SMP kernel 2.2.10. In our experiments, we are using a single processor per node.

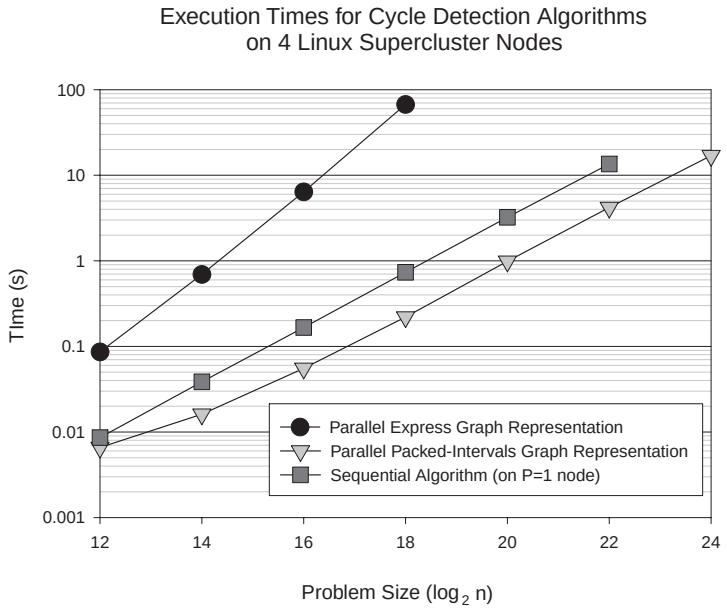

Figure 4: Comparison of the Sequential and Parallel Cycle Detection Algorithms.

In Figure 4 we compare the total running time for our parallel cycle detection algorithms on four processors with that of the best-known sequential algorithm. We use a log-log plot to capture the large range (several magnitudes) of input sizes and running times. Notice that in all cases, the best-known sequential approach is faster than the parallel Express algorithm. In addition, the largest problem that can be run for Express ($n = 2^{18}$ vertices) is 16 times smaller than the largest sequential problem that can be run on a single node ($n = 2^{22}$ vertices). On the other hand, the Packed-Intervals algorithm on four processors can process an input four times larger ($n = 2^{24}$ vertices) than the sequential approach, and Packed-Intervals exhibits an almost linear speedup in comparison with the sequential approach.

Next, we present performance graphs that are normalized to the running time per vertex. This normalization allows a comparison that spans a large range of graph sizes (from 1024 to more than 256 million vertices). For each algorithm, machine size, and problem size, the running time is split into the corresponding four steps: 1) computation required to find cycles on each local subgraph, 2) sorting and communication of the trans-arcs during the discovery phase, 3) construction of express graph in either the Express or Packed-Interval graph representation, and 4) the merging of the subgraphs.

Figure 5 shows the performance of our cycle detection of the lattice input, for (A) 4 processors, (B) 16

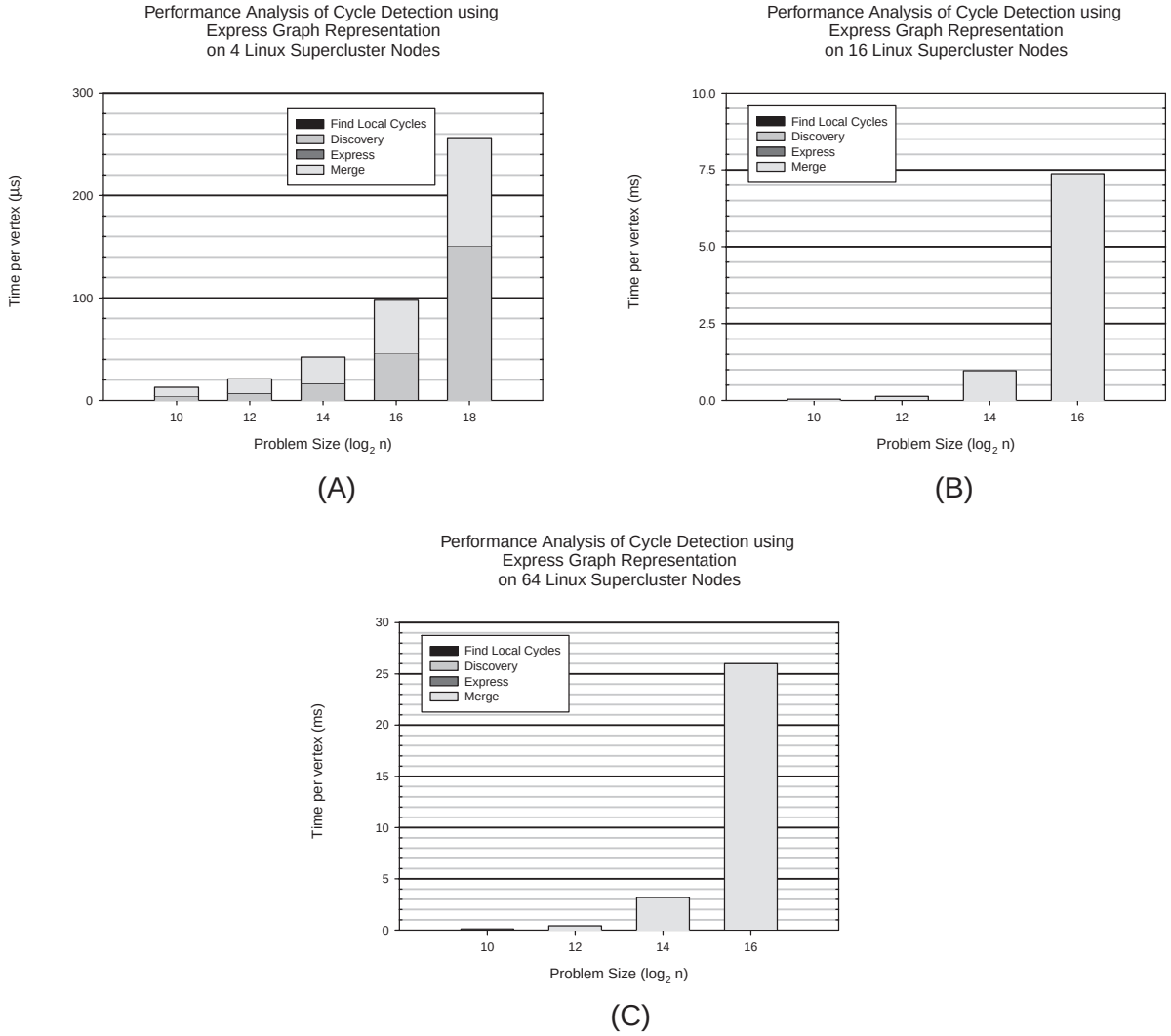

Figure 5: Performance of cycle detection using the Express graph representation

processors, and (C) 64 processors, and increasing problem sizes from 1024 vertices, to the largest problem that fits in memory, with each input four times larger than the last. Since we found a theoretical total complexity of $\tau \log p + O\left(\frac{n}{\sqrt{p}}\right)\sigma + O\left(\frac{n^2}{p^2}\right)$, the normalized $(1/n)$ running time is $\frac{\tau \log p}{n} + O\left(\frac{1}{\sqrt{p}}\right)\sigma + O\left(\frac{n}{p^2}\right)$. Thus, for fixed machine size p, and as our experimental results confirm, the normalized running time per vertex increases at a rate linear in the problem size n.

Figure 6 shows the corresponding performance for the Packed-Intervals graph representation, for (A) 1 processor, (B) 4 processors, (C) 16 processors, and (D) 64 processors, and increasing problem sizes from 2048 vertices, to the largest problem that fits in memory, with each input four times larger than the last. Plot 6(A) corresponds to the best sequential implementation for cycle detection and does not run any parallel steps. Since we found a theoretical total complexity of $\tau \log p + O\left(\frac{n}{p}\right)\sigma + O\left(\frac{n}{p}\right)$, the normalized $(1/n)$ running time is $\frac{\tau \log p}{n} + O\left(\frac{1}{p}\right)\sigma + O\left(\frac{1}{p}\right)$. Thus, for fixed p, and validated by the empirical performance, the normalized running time becomes constant as n increases. For small machine sizes, the discovery phase dominates the running time per vertex, but for large machine sizes, merging dominates. In addition, notice that in all cases, the running time per vertex is on the order of at most a few microseconds for Packed-Intervals, versus milliseconds for the

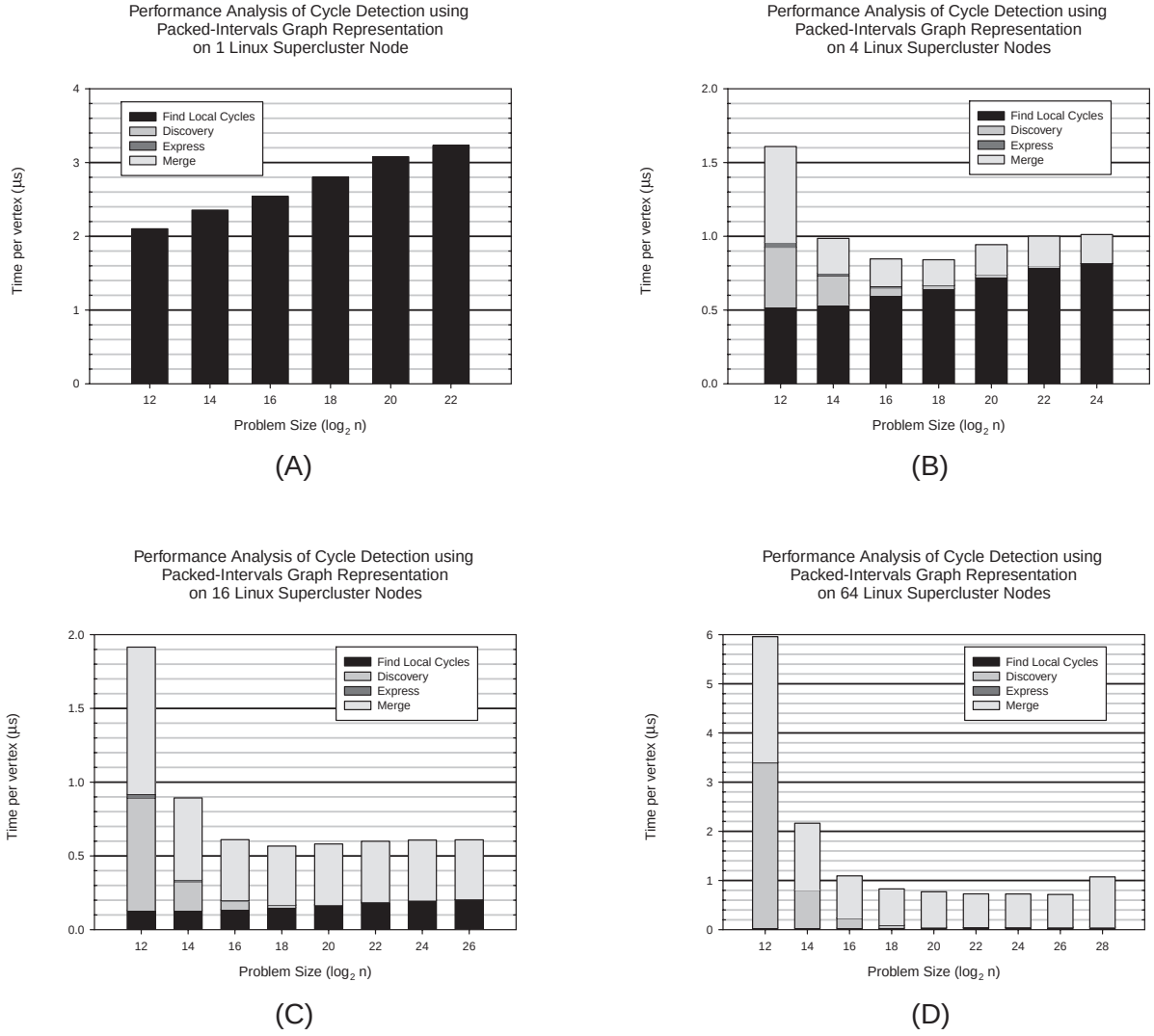

Figure 6: Performance of cycle detection using the Packed-Intervals graph representation

Express approach.

5 Discussion

We have discovered a new algorithm for efficiently detecting cycles in planar digraphs, using a compact data structure called the Packed-Intervals graph representation. We have demonstrated both through theoretical and practical analyses an algorithm that scales optimally with both machine and problem size, and in fact, is faster than the sequential algorithm for a problem of at least a thousand vertices. In addition, our new approach can solve large instances of graphs that previously could not be solved on current high-performance machines.

This technique can be applied to three-dimensional simulations that commonly detect cycles in planar cuts at various displacements and orientations. We also feel that the planar approach here may generalize to three-dimensional geometric graphs and will investigate further. We are also investigating other graph theoretic algorithms that may be significantly improved using the Packed-Intervals data structure.

References

[1] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One step closer towards a realistic model for parallel computation. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, Santa Barbara, CA, July 1995.

[2] D. A. Bader. *On the Design and Analysis of Practical Parallel Algorithms for Combinatorial Problems with Applications to Image Processing*. PhD thesis, University of Maryland, College Park, Department of Electrical Engineering, April 1996.

[3] D. A. Bader, A. B. Maccabe, J. R. Mastaler, J. K. McIver III, and P. A. Kovatch. Design and Analysis of the Alliance / University of New Mexico Roadrunner Linux SMP SuperCluster. In R. Buyya, M. Baker, K. Hawick, and H. James, editors, *Proceedings of the IEEE Computer Society International Workshop on Cluster Computing*, pages 9–18, Melbourne, Austrailia, December 1999.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[5] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[6] F. Glover. Maximum Matching in a Convex Bipartite Graph. *Naval Research Logistics Quarterly*, 14(3):313–316, 1967.

[7] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, New York, 1992.

[8] J. F. JáJá and K. W. Ryu. The Block Distributed Memory Model. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):830–840, August 1996.

[9] G. Karypis and V. Kumar. *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. Department of Computer Science, University of Minnesota, version 4.0 edition, September 1998.

[10] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Reinhart & Winston, New York, 1976.

[11] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, Inc., Mineola, NY, 1998.

[12] E. M. Reingold and R. E. Tarjan. On a Greedy Heuristic for Complete Matching. *SIAM Journal of Computing*, 10(4):676–681, 1981.

[13] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.