Chapter 2
Can Parallel Programming Revolutionize EDA Tools?

Yi-Shan Lu and Keshav Pingali

1 Introduction

I think this is the beginning of a beautiful friendship. Humphrey Bogart in *Casablanca*.

Until a decade ago, research in parallel programming was driven largely by the needs of computational science applications, which use techniques like the finite-difference and finite-element methods to find approximate solutions to partial differential equations. In finite differences, the key computational kernels are stencil computations on regular grids, and the solution of linear systems with structured sparsity such as banded systems. In finite-elements, the key computational kernel is the solution of sparse linear systems in which matrices have unstructured sparsity; these linear systems are usually solved using iterative methods like conjugate gradient in which the main computation is sparse matrix–vector multiplication.

Parallel programming research therefore focused largely on language support, compilation techniques, and runtime systems for matrix computations. Languages like High Performance FORTRAN (HPF) [21] and Coarray FORTRAN [32] were developed to make it easier to write matrix applications. Sophisticated compiler technology based on polyhedral algebra was invented to optimize loop nests that arise in matrix computations [6, 13, 14]. Runtime systems and communication libraries like OpenMP and MPI provided support for communication and synchronization patterns found in these applications.

While computational science applications and matrix computations continue to be important, our group at the University of Texas at Austin and several others across the world have shifted our focus to applications, such as the following ones, which compute on *unstructured graphs*.

Y.-S. Lu (✉) • K. Pingali
The University of Texas at Austin, Austin, TX, USA
e-mail: yishanlu@utexas.edu; pingali@cs.utexas.edu

© Springer International Publishing AG 2018

A.I. Reis, R. Drechsler (eds.), *Advanced Logic Synthesis*,
https://doi.org/10.1007/978-3-319-67295-3_2

- In social network analysis, the key data structures are extremely sparse graphs in which nodes represent entities and edges represent relationships between entities. Algorithms for breadth-first search, betweenness-centrality, page-rank, max-flow, etc. are used to extract network properties, to make friend recommendations, and to return results sorted by relevance for search queries [2, 20].
- Machine-learning algorithms like belief propagation and survey propagation are based on message-passing in a factor graph, which is a sparse bipartite graph [23].
- Data-mining algorithms like k-means and agglomerative clustering operate on dynamically changing sets and multisets [37].
- Traffic and battlefield simulations often use event-driven (discrete-event) simulation [27] in networks.
- Program analysis and instrumentation algorithms used within compilers are usually based on graphical representations of program properties, such as points-to graphs [3, 15, 25, 34].

Irregular graph applications such as these can have a lot of parallelism, but the patterns of parallelism in these programs are very different from the parallelism patterns one finds in computational science programs.

- Graphs in many of these applications are very dynamic data structures since their structure can be morphed by the addition and removal of nodes and edges during the computation. Matrices are not good abstractions for such graphs.
- Even if the graphs have fixed structure, many of the algorithms do not fit the matrix–vector/matrix–matrix multiplication computational patterns that are the norm in computational science. One example is delta-stepping, an efficient parallel single-source shortest-path (SSSP) algorithm [20, 26]. This algorithm maintains a work-list of nodes, partially sorted by their distance labels. Nodes enter and leave the work-list in a data-dependent, statically unpredictable order. This is a computational pattern one does not see in traditional computational science applications.
- Parallelism in irregular graph applications like delta-stepping is dependent not only on the input data but also on values computed at runtime. This parallelism pattern, which we call *amorphous* data-parallelism [33], requires parallelism to be found and exploited at runtime during the execution of the program. In contrast, the conventional data-parallelism in computational science kernels is independent of runtime values and can found by static analysis of the program.[1]

In spite of these difficulties, the parallel programming research community has made a lot of progress in the past 10 years in designing abstractions, programming models, compilers, and runtime systems for exploiting amorphous data-parallelism in graph applications. These advances have not yet had a substantial impact on EDA tools even though unstructured graphs underlie many EDA algorithms. The goal of

[1]Sparse direct methods are an exception, but even in these algorithms, a dependence graph, known as the elimination tree, is built before the algorithm is executed in parallel [5].

this paper is to summarize advances reported in previous papers [20, 33] and discuss their relevance to the EDA tools area, with the goal of promoting more interaction between the EDA tools and parallel programming communities.

The rest of this paper is organized as follows. Section 2 describes an abstraction for graph algorithms, called the operator formulation of algorithms. Section 3 discusses the patterns of parallelism in graph algorithms, and describes the Galois system, which exploits this parallelism while providing a sequential programming model implemented in C++. Section 4 summarizes the results of several case studies that use the Galois system, including scalability studies on large-scale shared-memory machines [18], and implementations of graph analytics algorithms [31], subgraph isomorphism algorithms, and FPGA maze routing [28]. We conclude in Sect. 5.

2 Abstractions for Graph Algorithms

Parallelism in matrix programs is usually described using program-centric concepts like parallel loops and parallel procedure calls. One lesson we have learned in the past 10 years is that parallelism in graph algorithms is better described using a *data-centric* abstraction called the *operator formulation of algorithms* in which data structures, rather than program constructs, play the central role [33]. To illustrate concepts, we use the single-source shortest-path (SSSP) problem. Given an undirected graph $G = (V, E, w)$ in which V is the set of nodes, E the set of edges, and w a map from edges to positive weights, the problem is to compute for each node the shortest distance from a source node s. There are many algorithms for solving this problem such as Dijkstra's algorithm, Bellman-Ford algorithm, delta-stepping and chaotic relaxation [20], but in the standard presentation, these algorithms appear to be unrelated to each other. In contrast, using the operator formulation elucidates their similarities and differences.

2.1 Operator Formulation

The operator formulation of an algorithm has a *local view* and a *global view*, shown pictorially in Fig. 2.1. This formulation of algorithms leads to a useful classification of algorithms, called TAO analysis, shown in Fig. 2.2.

2.1.1 Local View of Algorithms: Operators

The local view is described by an *operator*, which is a graph update rule applied to an *active node* in the graph (some algorithms have active *edges*, but to avoid verbosity, we refer only to active nodes in this paper). Each operator application,

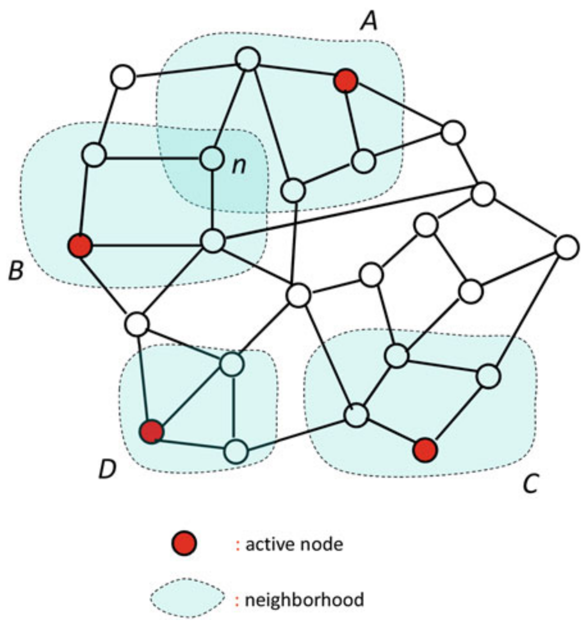


Fig. 2.1 Operator formulation

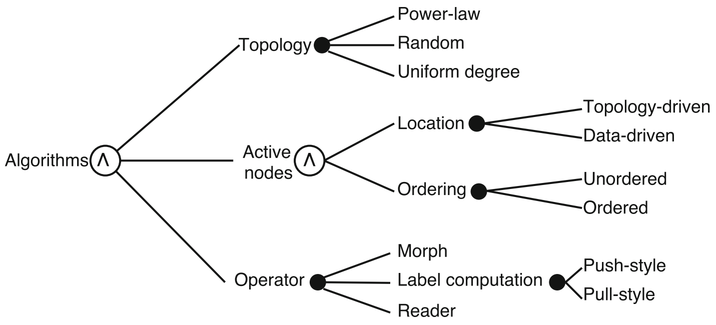


Fig. 2.2 TAO analysis of graph algorithms

called an *activity*, reads and writes a small region of the graph around the active node, called the *neighborhood* of that activity. In Dijkstra's SSSP algorithm, the operator, called the relaxation operator, uses the label of the active node to update the labels of its immediate neighbors. Figure 2.1 shows active nodes as filled dots, and neighborhoods as clouds surrounding active nodes, for a generic algorithm. An active node becomes inactive once the activity is completed.

In general, operators can modify the graph structure of the neighborhood by adding and removing nodes and edges (these are called *morph* operators). In most graph analytics applications, operators only update labels on nodes and edges, without changing the graph structure. These are called *label computation* operators; a *pull-style operator* reads the labels of nodes in its neighborhood and writes to the label of its active node, while a *push-style operator* reads the label of the active node and writes the labels of other nodes in its neighborhood. Dijkstra's algorithm uses a push-style operator. In algorithms that operate on several data structures, some data structures may be read-only in which case the operator is a *reader* for those data structures.

Neighborhoods can be distinct from the set of immediate neighbors of an active node, and in principle, can encompass the entire graph, although usually they are small regions of the graph surrounding the active node. Neighborhoods of different activities can overlap; in Fig. 2.1, node n is contained in the neighborhoods of both activities A and B. In a parallel implementation, the semantics of reads and writes to such overlapping regions, known as the *memory model*, must be specified carefully.

2.1.2 Global View of Algorithms: Location of Active Nodes and Ordering

The global view of a graph algorithm is captured by the location of active nodes and the order in which activities must appear to be performed.

Topology-driven algorithms make a number of sweeps over the graph until some convergence criterion is met; in each sweep, all nodes are active initially. The Bellman-Ford SSSP algorithm is an example. Data-driven algorithms, on the other hand, begin with an initial set of active nodes, and other nodes may become active on the fly when activities are executed. These algorithms do not make sweeps over the graph, and terminate when there is no more active nodes. Dijskstra's SSSP algorithm is a data-driven algorithm: initially, only the source node is active, and other nodes become active when their distance labels are lowered.

The second dimension of the global view of algorithms is *ordering*. In *unordered* algorithms, any order of processing active nodes is semantically correct; each sweep of the Bellman-Ford algorithm is an example. Some orders may be more efficient than others, so unordered algorithms sometimes assign *soft* priorities to activities, but these are only suggestions to the runtime system, and priority inversions are permitted in the execution. In contrast, *ordered* algorithms require that active nodes appear to have been processed in a specific order; Dijkstra's algorithm and algorithms for discrete-event simulation are examples. This order is specified by assigning priorities to active nodes, and the implementation is required to process active nodes so that they appear to have been scheduled for execution in *strict* priority order from earliest to latest.

2.2 Trade-offs Between Topology-Driven and Data-Driven Algorithms

Many graph problems, like SSSP, can be solved by both topology- and data-driven algorithms. However, one should be aware of the tradeoffs involved when choosing algorithms.

Topology-driven algorithms are easier to implement because iteration over active nodes can be implemented by traversing the nodes in the representation of the graph (usually arrays). However, there may be wasted work in each sweep because there may not be useful work done at many nodes.

In contrast, data-driven algorithms can be work-efficient since activities are performed where there is useful work to be done. For many problems, data-driven algorithms are asymptotically faster than topology-driven algorithms. For instance, the complexity of the Bellman-Ford algorithm is $O(|E||V|)$, whereas Dijkstra's algorithm is $O(|E|\log(|V|))$.

On the other hand, data-driven algorithms can be complicated to implement because they need a work-set to track active nodes. Sequential implementations use lists and priority queues for unordered and ordered algorithms, respectively. Concurrent work-lists for graph algorithms are difficult to implement efficiently: since the amount of work in each activity is usually fairly small, adding and removing active nodes from the work-list can become a bottleneck unless the work-list is designed very carefully. Nguyen et al. [19, 31] describe a scalable work-set called *obim* that supports soft priorities.

The best choice of algorithm for a given problem can also depend on the topology of the graph, as we show in Sect. 4.2 [9, 29]. In many social networks such as the web graph or the Facebook friends graph, the degree distribution of nodes roughly follows a power law, so these are often referred to as *power-law* graphs. In contrast, road networks and 2D/3D grids/meshes are known as *uniform-degree* graphs because most nodes have roughly the same degree. Graphs for VLSI circuits fall in this category. *Random* graphs, which are created by connecting randomly chosen pairs of nodes, constitute another category of graphs. Different graph classes have very different properties: for example, the diameter of a randomly generated power-law graph grows only as the logarithm of the number of nodes in the graph but for uniform-degree graphs, the diameter can grow linearly with the number of nodes [20].

Figure 2.2 summarizes this discussion. We call it TAO analysis for its three main dimensions: Topology of the input graph, Activity location and ordering, and Operator. Note that TAO analysis does not distinguish between sequential and parallel algorithms.

3 Exploiting Parallelism in Graph Algorithms

Parallelism can be exploited by processing active nodes in parallel, subject to neighborhood and ordering constraints. Since neighborhoods can overlap, the memory model, which defines the semantics of reads and writes in overlapped regions, may prevent some activities with overlapping neighborhoods from executing in parallel. In addition, ordering constraints between activities must be enforced. We call this pattern of parallelism *amorphous* data-parallelism [33]; it is a generalization of data-parallelism in which (1) there may be neighborhood and ordering constraints that

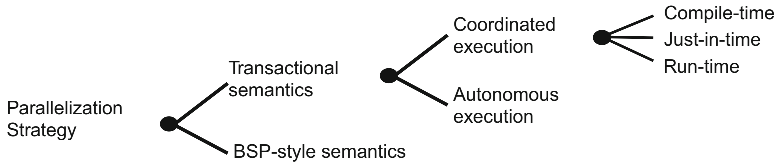


Fig. 2.3 Parallelization strategies

prevent all activities from executing in parallel, and (2) the execution of an activity may create new activities.

Figure 2.3 summarizes the important choices in implementing parallel graph programs. There are two popular memory models: BSP-style (Bulk-Synchronous Parallel) semantics [39] and transactional semantics.

3.1 BSP-Style Semantics

The program is executed in rounds (also known as super-steps), with barrier synchronization between rounds. Writes to the graph are considered to be communication from a round to the following round, so they are applied to the graph only at the beginning of the following round. Multiple updates to a label are resolved as in PRAM models such as by using a reduction operation to combine the updates into a single update [11].

BSP-style parallelization may work well for graph applications in which the number of activities in each round is large enough to keep the processors of the parallel machine busy. One example is breadth-first search (BFS) on a power-law graph. Each round handles nodes at a single BFS level and computes labels for nodes at the next BFS level. Since the average diameter of power-law graphs is small, there will be a lot of parallel activities in most rounds. On the other hand, BSP-style parallelization may not perform well for graphs that have high average diameter, such as road networks or VLSI circuits, as we show experimentally in Sect. 4.2. This is because the number of super-steps required to execute the algorithm may be large, and the number of activities in each super-step may be small.

3.2 Transactional Semantics

In this model, parallel execution of activities is required to produce the same answer as executing activities one at a time in some order that respects priorities. Intuitively, this means that activities should not "see" concurrently executing activities, and the updates made by an activity should become visible to other activities only after that activity completes execution. Formally, these two properties of transactional execution are known as isolation and atomicity.

Transactional semantics are implemented by preventing activities from executing in parallel if they *conflict*. For unordered algorithms, a conservative definition is that activities conflict if their neighborhoods overlap. In Fig. 2.1, activities A and B conflict because node n is in both their neighborhoods. Activities C and D do not conflict, and they can be executed in parallel with either A or B. Exploiting properties of the operator such as commutativity can lead to more relaxed definitions of conflicts, enhancing parallelism [16]. Given a definition of conflicts, the implementation needs to ensure that conflicting activities do not update the graph in parallel. This can be accomplished using *autonomous scheduling* or *coordinated scheduling*.

In autonomous scheduling, activities are executed speculatively. If a conflict is detected with other concurrently executing activities, some activities are aborted, enabling others to make progress; otherwise, the activity commits, and its updates become visible to other activities. Autonomous scheduling is good for exploiting parallelism but for some unordered algorithms, the output of the program can depend on the precise order in which activities are executed so the output may be non-deterministic in the sense that different runs of the program for the same input may produce different outputs. Delaunay mesh refinement and maze routing, discussed in Sect. 4.4, are examples. It is important to notice that this non-determinism, known as *don't-care non-determinism*, arises from under-specification of the order in which activities must be processed, and not from race conditions in updating shared state: even in a sequential implementation, the output of the program can depend on the order in which the work-list of active nodes is processed.

Coordinated scheduling strategies ensure that conflicting activities do not execute simultaneously [33]. For some algorithms, such as those that can be expressed using generalized sparse matrix–vector product, static analysis of the operator shows that active nodes can be executed in parallel without any conflict-checking. This is called *static parallelization*, and it is similar to auto-parallelization of dense array programs. *Just-in-time parallelization* preprocesses the input graph to find conflict-free schedules (e.g., by graph coloring); it can be used for topology-driven algorithms like Bellman-Ford in which neighborhoods are independent of data values. *Runtime parallelization* is general: the algorithm is executed in a series of rounds, and in each round, a set of active nodes is chosen, their neighborhoods are computed, and a set of non-conflicting activities are selected and executed. This approach can be used for deterministic execution of unordered algorithms [31].

3.3 The Galois System

The Galois system is an implementation of these data-centric abstractions.[2] Application programmers write programs in sequential C++, using certain programming

[2]A more detailed description of the implementation of the Galois system can be found in our previous papers such as [31].

```
1  #include "Galois/Galois.h"
2
3  struct Data {int dist;};
4  typedef Galois::Graph::LC_CSR_Graph<Data,void> Graph;
5  typedef Graph::GraphNode Node;
6
7  struct BFS {
8    Graph& g;
9    BFS(Graph& g) : g(g) {}
10   void operator()(Node n, Galois::UserContext<Node>& ctx) {
11     int newDist = g.getData(n).dist + 1;
12     for(auto e : g.edges(n)) {
13       Node dst = g.getEdgeDst(e);
14       int& dstDist = g.getData(dst).dist;
15       if(dstDist > newDist) {
16         dstDist = newDist;
17         ctx.push(dst);
18       }
19     }
20   }
21 };
22
23 int main(int argc, char** argv) {
24   Graph g;
25   Galois::readGraph(g, argv[1]);
26   Galois::do_all_local(g, [&g] (Node n) {g.getData(n).dist = DIST_INFINITY;});
27   int start = atoi(argv[2]);
28   Node src = *(std::advance(g.begin(),start));
29   g.getData(src).dist = 0;
30   Galois::for_each(src, BFS{g});
31   return 0;
32 }
```

Fig. 2.4 Push-style BFS in Galois

patterns, described below, to highlight opportunities for exploiting amorphous data-parallelism.

Key features of the system are described below, using the code for push-style BFS shown in Fig. 2.4. This code begins by reading a graph from a file (line 25) and constructing a compressed-sparse-row (CSR) representation in memory (line 4). Line 26 initializes the `dist` fields of all nodes to ∞, and lines 27–29 read in the ID of the source node and initialize its `dist` field to 0.

- Application programmers specify parallelism *implicitly* by using Galois set iterators [33] which iterate over a work-list of active nodes. For data-driven algorithms, the work-list is initialized with a set of active nodes before the iterator begins execution. The execution of a iteration can create new active nodes, and these are added to the work-list when that iteration completes execution. Topology-driven algorithms are specified by iteration over graph nodes, and the iterator is embedded in an ordinary (sequential) loop, which iterates until the convergence criterion is met. In Fig. 2.4, data-driven execution is specified by line 30, which uses a Galois set iterator to iterate over a work-list initialized to contain the source node `src`.
- The body of the iterator is the implementation of the operator, and it is an imperative action that reads and writes global data structures. In Fig. 2.4, the operator is specified in lines 10–20. This operator iterates over all the neighbors

of the active node, updating their distance labels as needed. Iterations are required to be *cautious*: an iteration must read *all* elements in its neighborhood before it writes to *any* of them [33]. In our experience, this is not a significant restriction since the natural way of writing graph analytics applications results in cautious iterations.

- For unordered algorithms, the relative order in which iterations are executed is left unspecified in the application code. An optional application-specific priority order for iterations can be specified with the iterator [30], and the implementation tries to respect this order when it schedules iterations.
- The system exploits parallelism by executing iterations in parallel. To ensure serializability of iterations, programmers must use a library of built-in concurrent data structures for graphs, work-lists, etc. These library routines expose a standard API to programmers, and they implement lightweight synchronization to ensure serializability of iterations, as explained below.

Inside the data structure library, the implementation of a data structure operation such as reading a graph node or adding an edge between two nodes acquires logical locks on nodes and edges before performing the operation. If the lock is already owned by another iteration, the iteration that invoked the operation releases all of its acquired locks and is rolled back; it is retried again later. Intuitively, the cautiousness of iterations reduces the synchronization problem to the dining philosopher's problem [4], obviating the need for more complex solutions like transactional memory. The system also supports BSP-style execution of activities, and this can be specified by the user using a directive for the iterator. This is useful for deterministic execution of unordered algorithms.

4 Using Galois: Case Studies

This section discusses a number of case studies in which the Galois system is used to parallelize algorithms from several domains. Section 4.1 shows the scalability of Galois programs for HPC and graph analytics algorithms on a large-scale NUMA shared-memory machine. Section 4.2 compares Galois program performance with the performance of programs in Ligra [36] and PowerGraph [7], two popular shared-memory graph analytics systems. We show that for road networks, which are high-diameter graphs like circuit graphs, Galois programs run orders of magnitude faster than Ligra and PowerGraph programs. Section 4.3 describes implementations of subgraph matching algorithms in Galois. Finally, Sect. 4.4 describes how the Galois system was used by Moctar and Brisk to perform parallel FPGA maze routing [28].

4.1 Case Study: Large-Scale Shared-Memory Machines

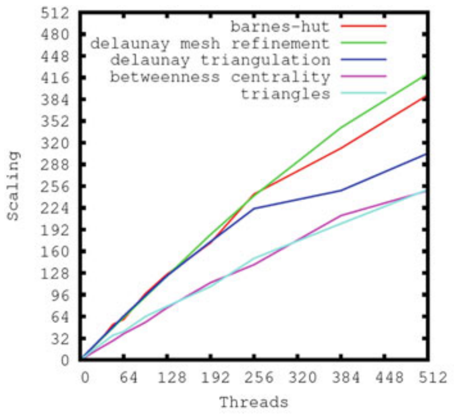
In this section, we summarize the results of a study by Lenharth and Pingali on the performance of Galois programs on a large-scale NUMA shared-memory machine [18]. The machine used in this study is the Pittsburgh Supercomputing Center's Blacklight system, which is an SGI UltraViolet NUMA system with 4096 cores and 32 TiB of ram (our machine allocation was limited to 512 cores). Each NUMA node contains 16 cores running at 2.27 GHz on two packages and 128 GiB of memory. We compile using g++ 4.7 at -O3. The benchmarks used are Barnes-Hut (`bh`), an n-body simulation code; Delaunay mesh generator (`dt`), a guaranteed-quality 2D triangular mesh generator; Delaunay mesh refinement (`dmr`), a mesh refinement algorithm for 2D meshes; betweenness centrality (`bc`), a centrality computation in networks; and triangle counting (`tri`), which counts the number of triangles in a graph. Table 2.1 summarizes the inputs and configurations used. Although these results are on the SGI machine, similar results are seen on smaller scale NUMA systems.

Figure 2.5 shows that `dmr` and `bh` achieve self-relative, strong scaling of 422× and 390×, respectively, at 512 threads. This equates to an 82% and 75% parallel efficiency for programs written in a sequential programming style. Delaunay triangulation scales only to 304× at 512 threads, due in part to memory contention when inserting into the lookup-acceleration tree. Betweenness centrality requires reading the entire graph in each iteration. Although the graph size is small enough to fit in the L3 cache, the temporary data necessary for an "outer-loop" parallel `bc` calculation is proportional to the size of the graph, so in actual parallel execution, the graph could not remain in cache. Adding NUMA nodes increases the number of cores but hurts the average latency of memory accesses for all threads, and this causes `bc` to scale at only about 50% efficiency. Triangle finding scales at about 50% efficiency.

Scaling numbers can be deceptive because they do not take into account the effect of single-thread overheads. Therefore, we also compare the single-threaded performance of the Galois codes to third-party serial implementations of these algorithms. Our goal is not necessarily to have the best performing serial implementation, especially since some of the implementations use hand-crafted,

Table 2.1 Inputs used in evaluation on SGI Ultraviolet

App	Input and configuration
`bh`	10 million bodies generated using a plummer model, tolerance = 0.05, timestep = 0.5, eps = 0.05
`dmr`	20 million triangles in a square, 50% bad
`dt`	10 million points randomly distributed in a square
`bc`	Random graph with average degree 4 and 2^{18} nodes
`tri`	Random planar graph with average degree 4 and 2^{28} nodes



App	Implementation	Threads	Time (s)
dmr	triangle	1	96
	Galois	1	155.7
	Galois	512	0.37
dt	triangle	1	1185
	Galois	1	56.6
	Galois	512	0.18
bh	splash2	1	>6000
	Galois	1	1386
	Galois	512	3.55
bc	HPCS SSCA	1	6720
	Galois	1	5394
	Galois	512	21.6
tri	graphlab	2	531
	Galois	1	7.03
	Galois	512	0.028

Fig. 2.5 Performance of Galois programs: SGI Ultraviolet

problem-specific data-structures, but to show that we are within an acceptable margin of custom implementations even on one thread while using the generic data-structures provided by our runtime.

Figure 2.5 shows that we compare favorably with third party implementations for all our benchmarks. For Delaunay triangulation and Delaunay mesh refinement, we compare to Triangle [35]. Our implementation of refinement on one thread is slower than Triangle, but triangulation is much faster (due to a more efficient algorithm). For Barnes-Hut, we compare to SPLASH-2 [40]. Although the SPLASH implementation is an ancestor of our implementation, ours is slightly faster. For betweenness centrality, we compare to the Scalable Synthetic Compact Applications benchmark suite [1] and we are 20% faster. Finally, compared to the nearly identical implementation in GraphLab [22], our implementation of triangles on 1 thread is 75× faster than Graphlab on 2 threads (the Graphlab code did not terminate when it was run on 1 thread).

4.2 Case Study: Graph Analytics

Parallel graph analytics has become a popular area of research in the past few years. In these applications, labels on nodes are repeatedly updated until some convergence criterion is reached, but the graph structure is not modified (in the TAO classification described in Fig. 2.2, these algorithms use label computation operators). Nguyen et al. [31] compared the performance of graph analytics applications written in Galois with the performance of the same applications written in two other frameworks, PowerGraph [7] and Ligra [36]. We summarize their study in this section.

PowerGraph [7] is a programming model for vertex programs, an abstraction in which the neighborhood of an active node is limited to itself and the set of its immediate neighbors [24]. It supports shared-memory parallelism in a single machine as well as distributed-memory execution on clusters. *Ligra* [36] is a shared-memory programming model for vertex programs. Ligra is capable of switching between pull- and push-style operators during execution time to improve cache utilization. Both PowerGraph and Ligra support only bulk synchronous parallelism (BSP).

Unlike most graph analytics studies, the study of Nguyen et al. used both power-law graphs (twitter-50 with 51 million nodes and 2 billion edges) and road networks (U.S. road network with 24 million nodes and 58 million edges). *Graphs of importance in the EDA tools area, such as circuit graphs, are likely to be high-diameter graphs similar to road networks, so this study sheds some light on what kinds of graph processing systems are likely to be useful for parallel EDA tools.*

Nguyen et al. used the following applications in their study:

Single-source shortest-paths (`SSSP`) is the problem used in Sect. 2 to illustrate the operator formulation of algorithms.

Breadth-first search (`BFS`) is a special case of `SSSP` in which all edge weights are one.

Approximate diameter (`DIA`) computes an approximation to the graph diameter, which is the maximum length of the shortest distance between all pairs of nodes.

Connected components (`CC`) divides the nodes of an undirected graph into equivalence classes by reachability.

Pagerank (`PR`) computes a relative importance score for each node in a graph.

Figure 2.6 compares the performance of the three graph analytics frameworks with different pairs of applications and input graphs. The experiments were run on a machine with 40-core Intel E7-4860 and 128 GB of memory. Notice that the y-axis is a log-scale.

Although the road network is roughly 40 times smaller than the Twitter graph, Ligra and PowerGraph take far more time for `BFS` and `SSSP` on the road network than on the Twitter graph. The U.S. road network has a large diameter and a uniform, low degree distribution, so BSP-style implementation of algorithms requires a large number of low-parallelism rounds. Galois avoids this problem by providing asynchronous scheduling of activities, and is orders of magnitude faster than Ligra and PowerGraph.

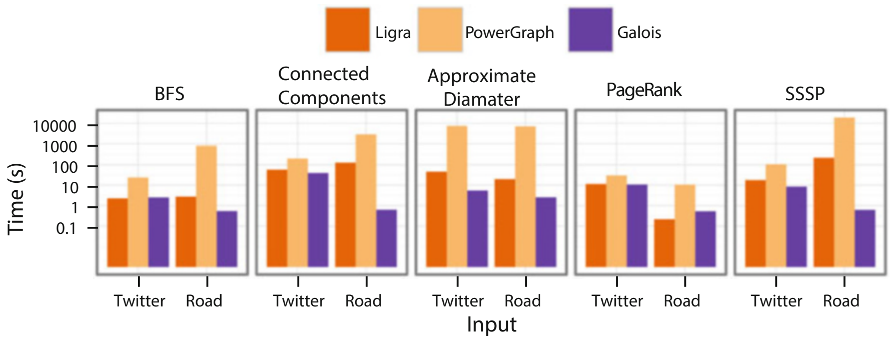


Fig. 2.6 Comparison of graph analytics frameworks [31]

The performance of `CC` highlights the value of Galois's support of operators with arbitrary neighborhoods. Since Ligra and PowerGraph support only vertex programs, `CC` on these systems requires a label-propagation algorithm: all nodes are given distinct component IDs, and the nodes with smallest IDs propagate their IDs to all nodes in their components. On the other hand, the Galois program uses a parallel union-find data structure. The union-find data structure is updated by pointer jumping, which is similar to the find operation in disjoint-set union-find [11, 12]. However, pointer jumping cannot be expressed as a vertex program, so the program cannot be written using Ligra and PowerGraph.

In summary, runtimes vary widely for different programming models even for the same problem. Pagerank demonstrates the least variation since all three frameworks use a topology-driven algorithm. For other problems, systems like Ligra and PowerGraph that are based on BSP-style execution of vertex programs can be several orders of magnitude slower than Galois, especially for road networks.

4.3 *Case Study: Subgraph Isomorphism*

Subgraph isomorphism is an important kernel for many applications; for example, it can sometimes be used to locate and then replace a suboptimal circuit with a functionally equivalent circuit with better delay, area, or power consumption. Formally, subgraph isomorphism is defined as follows [38]: given a query graph $G^q = (V^q, E^q)$ and a data graph $G^d = (V^d, E^d)$, a subgraph isomorphism exists between G^q and G^d if and only if there exists a function $I : V^q \to V^d$ such that

1. $i \neq j \implies I(v_i^q) \neq I(v_j^q)$, and
2. $(v_i^q, v_j^q) \in E^q \implies (I(v_i^q), I(v_j^q)) \in E^d$.

Intuitively, the function I, which maps query graph nodes to data graph nodes, is (1) injective, and (2) if two query nodes are connected by an edge, their images in the data graph are also connected by an edge. This definition assumes unlabeled graphs; it can be extended to deal with graphs with labeled nodes and edges. Subgraph isomorphism can be solved trivially by a generate-and-test approach in which all possible injective functions mapping query graph nodes to data graph nodes are generated, and each one is tested to see if it satisfies condition (2). To make this approach tractable, we must avoid generating, whenever possible, mappings that fail the test. There are many heuristics for this, and they can be described abstractly by the template shown in Algorithm 1 [17].

Procedure `GenericGraphQuery` preprocesses the graph by determining for each query node, an initial set of candidate data nodes that it can be mapped to, based on properties such as node labels, degrees, etc. If any query node has an empty set of candidate data nodes, then no subgraph isomorphism can be found. Otherwise, we call procedure `SubgraphSearch` to compute all subgraph isomorphisms, starting from an empty matching M.

Algorithm 1 Generic graph query algorithm for finding subgraph isomorphism

```
Input: $G^q$, query graph; $G^d$, data graph.
Output: All $I : V^q \rightarrow V^d$ satisfying subgraph isomorphism.
 1: procedure GenericGraphQuery($G^q, G^d$)
 2:     for all $v_i^q \in G^q$ do
 3:         $C_{v_i^q} \leftarrow$ FilterCandidates($v_i^q, G^q, G^d, \ldots$)
 4:         if $C_{v_i^q} = \emptyset$ then
 5:             return
 6:         end if
 7:     end for
 8:     $M \leftarrow \emptyset$
 9:     SubgraphSearch($G^q, G^d, M, \ldots$)
10: end procedure
11: procedure SubgraphSearch($G^q, G^d, M, \ldots$)
12:     if $|M| = |V^q|$ then
13:         report $V^q \rightarrow M$ in matching order
14:         return
15:     end if
16:     $v_j^q \leftarrow$ NextQueryNode($G^q, G^d, M \ldots$)
17:     $R \leftarrow$ RefineCandidates($v_j^q, G^q, C_{v_j^q}, G^d, M \ldots$)
18:     for all $v_k^d \in R$ do
19:         if IsJoinable($v_j^q, G^q, v_k^d, G^d, M \ldots$) then
20:             update $M$
21:             SubgraphSearch($G^q, G^d, M \ldots$)
22:             restore $M$
23:         end if
24:     end for
25: end procedure
```

In Procedure `SubgraphSearch`, the call to `NextQueryNode` heuristically chooses v_j^q, the next query node to be matched. Next, procedure `RefineCandidates` refines R, the set of candidate data nodes for v_j^q, based on current matching. Finally, for every data node v_k^d in R, procedure `IsJoinable` checks if v_k^d can be joined to current matching M and can satisfy all edge constraints. If so, we add v_k^d to M, recurse on `SubgraphSearch`, and remove v_k^d from M when the recursive call terminates.

Algorithms for subgraph isomorphism differ in the design of procedures `FilterCandidates`, `NextQueryNode`, and `RefineCandidates`.

- The *Ullmann* algorithm [17] constructs initial sets of candidate data nodes based on node labels, follows input order for deciding next query node to be matched, and does no refinement for sets of candidate data nodes in `SubgraphSearch`.

- The *VF2* algorithm [17] starts search from the first query node. It considers only unmatched immediate neighbors of matched query/data nodes when choosing next query node and refining sets of candidate data nodes. Also, a candidate data node is discarded by `RefineCandidates` if its number of unmatched neighbors is fewer than that of its query counterpart.

To parallelize these two algorithms, we view the search process as traversing a search tree whose nodes are query nodes being matched and branches (edges) are data nodes matched to the corresponding query nodes. Different subtrees share no information with each other, and the search process does not modify the query and data graphs. Therefore, we parallelize Ullmann and VF2 algorithms along the first level of the underlying search tree. Each task is a pair consisting of the first query node and one of its candidate data nodes. Threads claim tasks from a work queue, which is populated with all tasks initially. Using this parallelization strategy, Ullmann and VF2 algorithms are unordered, data-driven algorithms using reader operators.

The Ullmann and VF2 algorithms are implemented using Galois 2.3.0 beta with Boost library ver. 1.58.0. Experiments are run on a Linux server with 32 cores of Intel Xeon E7520 running at 1.87 GHz and 64GB RAM. We use two data graphs to run our experiments. RoadNY is the road network of New York City, and THEIA is a graph extracted from operating system activity logs. Table 2.2 summarizes the statistics for the data graphs. Figure 2.7 shows the query graph used in our experiments. All data graphs and the query graph are directed and unlabeled.

Figure 2.8 shows performance numbers. For roadNY, the Ullmann algorithm scales well but the VF2 algorithm outperforms it significantly. Since the sizes of frontiers, i.e. the number of unmatched immediate neighbors of matched nodes, grow slowly in road networks, filtering by frontiers works well. The scaling for VF2 is not as good as the scaling for Ullmann since the graphs are small. For THEIA, VF2 still outperforms Ullmann but not as much as when the data graph is roadNY.

Table 2.2 Data graphs for subgraph isomorphism

Graph	$\lvert V\rvert$	$\lvert E\rvert$	Est. diameter
roadNY	264,346	730,100	720
THEIA	7,478	17,961	2

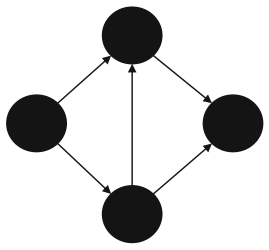


Fig. 2.7 The query graph for subgraph isomorphism

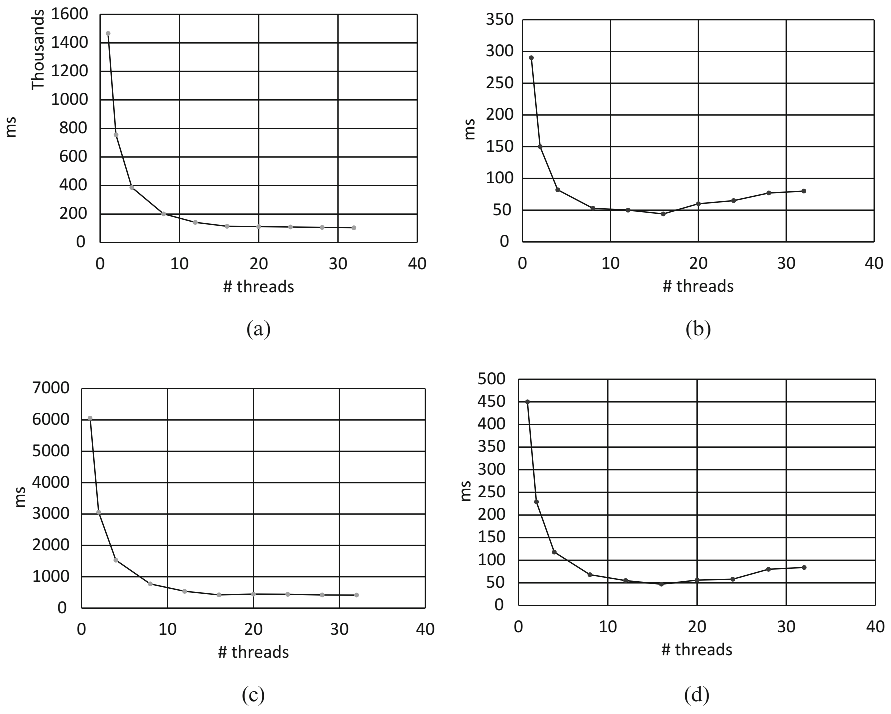
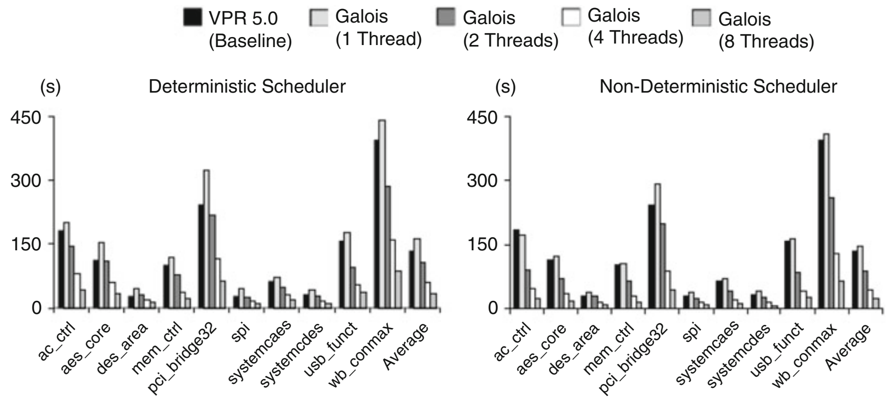


Fig. 2.8 Performance for finding 100 instances of the query graph in data graphs; (**a**) in roadNY by ullmann, (**b**) in roadNY by vf2, (**c**) in THEIA by ullmann, (**d**) in THEIA by vf2

Since THEIA is more similar to a power-law graph, its frontier grows rapidly, a fact which renders VF2's filtering by frontiers less effective in reducing the number of possible candidates to match.

4.4 Case Study: Maze Routing in FPGAs

Moctar and Brisk have used the Galois system to parallelize PathFinder, the routing algorithm used in many commercial FPGA tool chains [28]. Previous studies [8] have shown that roughly 70% of the execution time of PathFinder is spent in *maze expansion*, which performs an A* search of a routing resource graph (RRG) to route signals through the chip. This search procedure can be parallelized, but if different nets end up sharing routing resources, the resulting solution is illegal. When this happens, PathFinder needs to back up and reroute some nets to restore legality. PathFinder fails if a legal solution is not discovered within a user-specified number of iterations.



Averages	Deterministic Scheduler		Non-Deterministic Scheduler	
	VPR 5.0	134.6 seconds	VPR 5.0	134.6 seconds
	Galois (1 Thread)	162.4 seconds	Galois (1 Thread)	145.3 seconds
	Galois (2 Threads)	106.6 seconds	Galois (2 Threads)	88.8 seconds
	Galois (4 Threads)	59.2 seconds	Galois (4 Threads)	43.0 seconds
	Galois (8 Threads)	33.7 seconds	Galois (8 Threads)	22.6 seconds

Fig. 2.9 Performance of Maze Router in Galois [28]

This computation pattern maps well to the asynchronous optimistic execution model described in Sect. 3. Moctar and Brisk evaluated their implementation on ten of the largest IWLS benchmarks [10], using an 8-core Intel Xeon platform. As a baseline, circuits were generated for these benchmarks using the publicly available Versatile Placement and Routing (VPR) system. A rough idea of the size of these circuits can be obtained from the fact that one of the largest circuits, generated for the `wb_conmax` input, had 10,430 nets and 6,297 logic blocks. Key findings from the study include the following.

- The Galois implementation was able to successfully route all of the nets in all benchmarks.
- The average speed-up over the serial VPR implementation was roughly 3 for 4 threads, and 5.5 for 8 threads, as shown in Fig. 2.9.
- The implementation was roughly three times faster than the best previous implementation of parallel multi-threaded FPGA routing.
- Because of the don't-care non-determinism in autonomous scheduling, different runs of the Galois program can produce different routing solutions, but the variation in critical path delay was small.

To eliminate don't-care non-determinism, the study also used the deterministic, round-based execution of Galois programs described in Sect. 3. This reduced speed-up to 4 on 8 threads, but ensured that each run of the program, even on different numbers of cores, returned the same routing solution.

Moctar and Brisk summarized their results as follows: "we *strongly believe* that Galois' approach is the *right* solution for parallel CAD, due to the widespread use of graph-based data structures (e.g. netlists) that exhibit irregular parallelism" [28].

5 Conclusions

In the past decade, there has been a lot of progress in understanding the structure of parallelism in graph computations, and there is now a rich literature on programming notations, compilers, and runtime systems for large-scale graph computations. Meanwhile, circuit designs have become sufficiently complex that the EDA tools community has begun to look at parallel computation as a way of speeding up the design process. Therefore the time has come for closer interaction between the EDA tools and parallel programming communities. We hope this paper catalyzes this interaction.

Acknowledgements This research was supported by NSF grants 1218568, 1337281, 1406355, and 1618425, and by DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563.

References

1. D.A. Bader, K. Madduri, Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors, in *High Performance Computing, HiPC'05* (2005)
2. U. Brandes, T. Erlebach (eds.), *Network Analysis: Methodological Foundations* (Springer, Heidelberg, 2005)
3. G. Bronevetsky, D. Marques, K. Pingali, P. Stodghill, C3: a system for automating application-level checkpointing of MPI programs. *Languages and Compilers for Parallel Computing* (Springer, New York, 2004), pp. 357–373
4. K.M. Chandy, J. Misra, The drinking philosophers problem. ACM Trans. Program. Lang. Syst. **6**(4), 632–646 (1984)
5. I.S. Duff, A.M. Erisman, J.K. Reid, *Direct Methods for Sparse Matrices* (Oxford University Press, New York, 1986). ISBN 0-198-53408-6
6. P. Feautrier, Some efficient solutions to the affine scheduling problem: one dimensional time. Int. J. Parallel Prog. **21**(5), 313–347 (1992)
7. J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: distributed graph-parallel computation on natural graphs, in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12*, Berkeley, CA, 2012, pp. 17–30. USENIX Association. ISBN 978-1-931971-96-6. http://dl.acm.org/citation.cfm?id=2387880.2387883
8. M. Gort, J. Anderson, Deterministic multicore parallel routing for FPGAs, in *International Conference on Field Programmable Technology (ICFPT'10)* (2010)
9. M.A. Hassaan, M. Burtscher, K. Pingali, Ordered vs unordered: a comparison of parallelism and work-efficiency in irregular algorithms, in *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming, PPoPP '11* (ACM, New York, 2011), pp. 3–12. ISBN 978-1-4503-0119-0. doi:http://doi.acm.org/10.1145/1941553.1941557. http://iss.ices.utexas.edu/Publications/Papers/ppopp016s-hassaan.pdf
10. IWLS, IWLS 2005 benchmarks. http://iwls.org/iwls2005/benchmarks.html (2005)
11. J. JaJa, *An Introduction to Parallel Algorithms* (Addison-Wesley, Boston, 1992)
12. R.M. Karp, V. Ramachandran, A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD-88-408, EECS Department, University of California, Berkeley (1988). http://www.eecs.berkeley.edu/Pubs/TechRpts/1988/5865.html
13. K. Kennedy, J. Allen (eds.) *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach* (Morgan Kaufmann, San Francisco, 2001)

14. I. Kodukula, N. Ahmed, K. Pingali, Data-centric multi-level blocking, in *PLDI '97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (ACM, New York, 1997), pp. 346–357. ISBN 0-89791-907-6. doi:http://doi.acm.org/10.1145/258915.258946. http://iss.ices.utexas.edu/Publications/Papers/PLDI1997.pdf
15. V. Kotlyar, K. Pingali, P. Stodghill, A relational approach to the compilation of sparse matrix programs, in *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing* (Springer, London, 1997), pp. 318–327. ISBN 3-540-63440-1. http://iss.ices.utexas.edu/Publications/Papers/EUROPAR1997.pdf
16. M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, K. Pingali, Exploiting the commutativity lattice, in *PLDI* (2011)
17. J. Lee, W.-S. Han, R. Kasperovics, J.-H. Lee, An in-depth comparison of subgraph isomorphism algorithms in graph databases. Proc. VLDB Endow. **6**(2), 133–144 (2012). ISSN 2150-8097. doi:10.14778/2535568.2448946. http://dx.doi.org/10.14778/2535568.2448946
18. A. Lenharth, K. Pingali, Scaling runtimes for irregular algorithms to large-scale NUMA systems. Computer **48**(8), 35–44 (2015)
19. A. Lenharth, D. Nguyen, K. Pingali, Priority queues are not good concurrent priority schedulers, in *European Conference on Parallel Processing* (Springer, Berlin/Heidelberg, 2015), pp. 209–221
20. A. Lenharth, D. Nguyen, K. Pingali, Parallel graph analytics. Commun. ACM **59**(5), 78–87 (2016). ISSN 0001-0782. doi:10.1145/2901919. http://doi.acm.org/10.1145/2901919
21. D.B. Loveman, High performance fortran. IEEE Parallel Distrib. Technol. Syst. Appl. **1**(1), 25–42 (1993)
22. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J.M. Hellerstein, Graphlab: a new parallel framework for machine learning, in *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010
23. D. Mackay, *Information Theory, Inference and Learning Algorithms* (Cambridge University Press, Cambridge, 2003)
24. G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing - "abstract", in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, PODC '09* (ACM, New York, 2009), pp. 6–6 ISBN 978-1-60558-396-9. doi:10.1145/1582716.1582723. http://doi.acm.org/10.1145/1582716.1582723
25. M. Mendez-Lojo, A. Mathew, K. Pingali, Parallel inclusion-based points-to analysis, in *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*, October 2010. http://iss.ices.utexas.edu/Publications/Papers/oopsla10-mendezlojo.pdf
26. U. Meyer, P. Sanders, Delta-stepping: a parallel single source shortest path algorithm, in *Proceedings of the 6th Annual European Symposium on Algorithms, ESA '98* (Springer, London, 1998), pp. 393–404. ISBN 3-540-64848-8. http://dl.acm.org/citation.cfm?id=647908.740136
27. J. Misra, Distributed discrete-event simulation. ACM Comput. Surv. **18**(1), 39–65 (1986), ISSN 0360-0300. doi:http://doi.acm.org/10.1145/6462.6485
28. Y.O.M. Moctar, P. Brisk, Parallel FPGA routing based on the operator formulation, in *Proceedings of the 51st Annual Design Automation Conference, DAC '14* (2014).
29. R. Nasre, M. Burtscher, K. Pingali, Data-driven versus topology-driven irregular computations on GPUs, in *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium, IPDPS '13* (Springer, London, 2013)
30. D. Nguyen, K. Pingali, Synthesizing concurrent schedulers for irregular algorithms, in *Proceedings of International Conference Architectural Support for Programming Languages and Operating Systems, ASPLOS '11*, pp. 333–344 (2011). ISBN 978-1-4503-0266-1. doi:10.1145/1950365.1950404. http://doi.acm.org/10.1145/1950365.1950404

31. D. Nguyen, A. Lenharth, K. Pingali, A lightweight infrastructure for graph analytics, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13* (ACM, New York, 2013), pp. 456–471. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522739. http://doi.acm.org/10.1145/2517349.2522739
32. R.W. Numrich, J. Reid, Co-array fortran for parallel programming, in *ACM Sigplan Fortran Forum*, vol. 17 (ACM, New York, 1998), pp. 1–31
33. K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M.A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, X. Sui, The TAO of parallelism in algorithms, in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, pp. 12–25 (2011). ISBN 978-1-4503-0663-8. doi:10.1145/1993498.1993501. http://doi.acm.org/10.1145/1993498.1993501
34. D. Prountzos, R. Manevich, K. Pingali, K.S. McKinley, A shape analysis for optimizing parallel graph programs, in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11* (ACM, New York, 2011), pp. 159–172. ISBN 978-1-4503-0490-0. doi:http://doi.acm.org/10.1145/1926385.1926405. http://www.cs.utexas.edu/users/dprountz/popl2011.pdf
35. J.R. Shewchuk, Triangle: engineering a 2D quality mesh generator and delaunay triangulator, in *Applied Computational Geometry: Towards Geometric Engineering*, ed. by M.C. Lin, D. Manocha. Lecture Notes in Computer Science, vol. 1148 (Springer, Berlin, 1996), pp. 203–222. From the First ACM Workshop on Applied Computational Geometry
36. J. Shun, G.E. Blelloch, Ligra: a lightweight graph processing framework for shared memory, in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, pp 135–146 (2013)
37. P.-N. Tan, M. Steinbach, V. Kumar (eds.), *Introduction to Data Mining* (Pearson Addison Wesley, Boston, 2005)
38. J.R. Ullmann, Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. J. Exp. Algorithm. **15**, 1.6:1.1–1.6:1.64 (2011) ISSN 1084-6654. doi:10.1145/1671970.1921702. http://doi.acm.org/10.1145/1671970.1921702
39. L.G. Valiant, A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990). ISSN 0001-0782. doi:http://doi.acm.org/10.1145/79173.79181
40. S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The splash-2 programs: characterization and methodological considerations, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95* (ACM, New York, 1995), pp. 24–36. ISBN 0-89791-698-0. doi:10.1145/223982.223990. http://doi.acm.org/10.1145/223982.223990