Kinetic Dependence Graphs *

M. Amber Hassaan
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
m.a.hassaan@utexas.edu

Donald Nguyen Keshav Pingali
Dept. of Computer Science
The University of Texas at Austin
ddn@cs.utexas.edu, pingali@cs.utexas.edu

Abstract

Task graphs or dependence graphs are used in runtime systems to schedule tasks for parallel execution. In problem domains such as dense linear algebra and signal processing, dependence graphs can be generated from a program by static analysis. However, in emerging problem domains such as graph analytics, the set of tasks and dependences between tasks in a program are complex functions of runtime values and cannot be determined statically. In this paper, we introduce a novel approach for exploiting parallelism in such programs. This approach is based on a data structure called the *kinetic dependence graph* (KDG), which consists of a dependence graph together with update rules that incrementally update the graph to reflect changes in the dependence structure whenever a task is completed.

We have implemented a simple programming model that allows programmers to write these applications at a high level of abstraction, and a runtime within the Galois system [15] that builds the KDG automatically and executes the program in parallel. On a suite of programs that are difficult to parallelize otherwise, we have obtained speedups of up to 33 on 40 cores, out-performing third-party implementations in many cases.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Concurrent programming structures.
Keywords: ordered algorithms; kinetic dependence graph; stable-source and unstable-source algorithms;

1. Introduction

A task graph is a directed acyclic graph (DAG) in which nodes represent computational tasks that must be performed atomically and edges represent dependences between tasks. A node without any predecessors in the task graph is called a *source*, and the corresponding task can be scheduled for execution on any free processor. When a task has completed, the corresponding node and its incident edges are removed from the graph, potentially creating new sources. Program execution terminates when the task graph becomes empty.

In some application areas such as dense numerical linear algebra, task graphs can be constructed statically because tasks and dependences are independent of runtime data values. Restructuring compilers can sometimes build the task graph automatically by analyzing the text of the program [23]. Static analyses can be fragile, so systems like DAGuE [38, 39], Intel CnC [8] and TBB [20] allow the programmer to explicitly specify tasks and dependences. Other systems like SMPSs, OpenMP 4.0, and XKaapi [16, 32, 33] require the programmer to specify the read and write sets (*i.e.* accesses to elements of shared data) of each task, and the system builds the task graph using this information. Finally, fork-join style programming models like Cilk [7, 10] implicitly specify a task graph through the parent-child relationship between a task and its children.

Conventional task graphs are not adequate for many applications, particularly in problem domains like graph analytics, time-based simulations and unstructured mesh computations. Applications considered in this paper include (i) asynchronous variational integrators (AVI), which are used in computational mechanics and in graphics for solving complicated contact mechanics problems [27] like simulating the tying of ribbons and the crushing of rabbits in a trash compactor [1, 18], (ii) discrete-event simulation (DES) [9], (iii) the simulation of collisions between a set of moving bodies like billiard balls [2, 18], and (iv) graph analytics algorithms such as Breadth-First Search (BFS), Minimum-weight Spanning Tree (MST), and tree traversals. The inadequacy of conventional task graphs for these applications arises from the fact that the dependence graph computed from a particular program state may not remain valid after a task has executed because new tasks may have been created and dependences between existing tasks may have changed. In particular, the following issues may arise upon executing a task.

* The work presented in this paper has been supported by the National Science Foundation grants CNS 1111766, CCF 1218568, XPS 1337281, and CNS 1406355.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey..

http://dx.doi.org/10.1145/2694344.2694363

1. A new task may be created that must be executed before some or all of the pending tasks. The execution order of tasks in these applications is problem-specific and input dependent. One example is Discrete-Event Simulation (DES), in which events have time-stamps based upon simulation time, and must appear to have been processed in time-stamp order to preserve causality. Executing tasks in task creation order or parent-child order, as is done in existing systems, may be incorrect.
2. Dependences between pending tasks may have changed. This happens when read and write sets of a task depend on values modified by execution of another task. Examples discussed in this paper include Kruskal's minimum-weight spanning tree algorithm and sparse matrix factorization. Most existing systems do not support this feature.
3. An incoming edge may be created to a source in the dependence graph. This may result from either of the behaviors mentioned above. This implies that it is not always safe to schedule multiple source from the dependence graph in parallel. We call algorithms with this property *unstable-source*. Examples include DES and asynchronous algorithms for simulating collisions.

These issues are not a concern in sequential implementations of these algorithms because they typically use a priority queue to organize tasks and execute tasks one at a time in priority order. For example, sequential DES implementations process events in increasing time order by maintaining a priority queue of events sorted by their time-stamp; it can be shown that it is correct to process the earliest event, which is a safe source in the DAG. These kinds of algorithms are called *ordered* algorithms in the TAO classification of algorithms [34] because the execution of the program must respect the priority order between tasks (in contrast, tasks in *unordered* algorithms can be executed in any order without violating program semantics)[1].

Some of these applications have been parallelized previously by other researchers using problem-specific techniques: for example, AVI has been parallelized by Huang *et al.* [19] using "edge-flipping" DAGs, described in Sections 2.1 and 4.1, but this technique cannot be used for any of the other problems considered in this paper. In the Galois project, we have shown that optimistic or speculative parallelization [11, 36] can be used effectively for unordered algorithms like Delaunay mesh refinement [31]; for ordered algorithms, the overheads of optimistic parallelization appear to outweigh benefits, as we show in this paper.

In this paper, we present a generalization of the task graph called the *kinetic dependence graph* (KDG), and show that it can be used to systematically parallelize ordered algorithms in a problem independent way. Problem-specific techniques used in the literature to parallelize these applications emerge as optimizations of the general KDG-based approach that exploit problem structure to reduce KDG overheads, and can easily be incorporated into our general approach. For example, in AVI, all sources in the task graph are guaranteed to be safe, which simplifies the implementation. We call such algorithms *stable-source*.

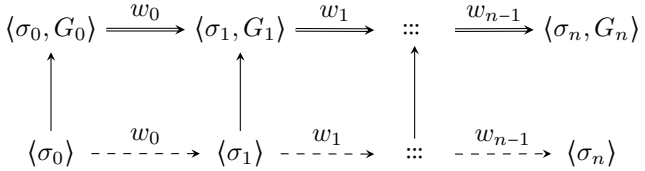


Figure 1: Kinetic dependence graph approach

Figure 1 is a pictorial representation of how the KDG is used for parallelization. A KDG has three components: (i) a task graph, (ii) a predicate called the *safe source test* for finding tasks that can be safely executed in parallel, and (iii) an *update rule* that specifies how the task graph should be updated incrementally when a task is completed. Given a program and an initial program state σ_0, we build the initial task graph G_0 at runtime, use the safe source test to select a task w_0, and execute that task to produce the new program state σ_1. Applying the (incremental) update rule to G_0 produces the new task graph G_1, which is the task graph that would have been produced from state σ_1, had we rebuilt the task graph from scratch[2]. Program execution terminates when the task graph becomes empty. The safe source test can be used to find multiple tasks that can be performed safely in parallel; the underlying system ensures that the resulting execution is serializable, i.e., the final state is equivalent to the state that would have been produced by executing these tasks serially. Conventional parallelization using task graphs is a special case of the KDG approach in which all sources of the task graph are safe sources, and the update rule just deletes the completed task and its associated edges from the task graph.

We have implemented KDGs for shared-memory machines within the Galois system [15]. Application programmers write sequential descriptions of ordered algorithms, while the Galois system builds and updates the KDG at runtime, yielding a correct and efficient parallelization. The programming model for these sequential descriptions is explained in Section 3. Our experimental results show speedups of up to 33 on a 40 core platform, demonstrating that the KDG approach can deliver substantial speedups from programs written at a high level of abstraction.

The rest of this paper is organized as follows. In Section 2, we introduce the KDG informally using AVI as the

[1] Ordered and unordered algorithms are similar to hyperbolic and elliptic partial differential equations respectively in the theory of PDEs [22].

[2] Strictly speaking, G_1 can be conservative in the sense that it may have more dependence edges.

```
1  Mesh M
2
3  struct Update {
4    Element* elem; double t; // time stamp;
5  };
6  WorkList<Update> Q;
7  for(Element* e: M.elements())
8    Q.push(Update(e, e->timestamp))
9
10 foreach (Update up ∈ Q) orderedby up.t:
11   Vec initial;
12   for (Vertex v: M.vertices(up.elem)):
13     initial.push(v.data);
14   Vec updated = AVI_update(up.elem, initial);
15   int i = 0;
16   for (Vertex v: M.vertices(up.elem)):
17     v.data = updated[i++];
18
19   up.elem->timestamp += up.elem->step;
20
21   if(up.elem->timestamp < endtime)
22     Q.push(Update(up.elem, up.elem->timestamp));
```

Figure 2: Asynchronous Variational Integrator (simplified)

running example. In Section 3, we describe the programming model and the KDG formally. We also give general procedures that can be used to construct the KDG for any application in this programming model, and show how to use the KDG for parallelization. In Section 3.6, we describe optimized KDG versions that can be used for applications with special structure. In Section 4, we describe the rest of the ordered applications considered in this paper. Experimental results are presented in Section 5. In these results, we compare the performance of the KDG approach with the performance of third-party implementations for our benchmark problems, if they are available. Related work is discussed in Section 6.

2. Informal introduction to KDGs

In this section, we use asynchronous variational integrators (AVI) to motivate and introduce the KDG.

2.1 AVI: Example Application

AVI is a finite-element method for solving PDEs. The domain of interest is discretized into a mesh of elements (triangles), such as the mesh shown in the Figure 3(a). When an element is updated, values stored at its vertices, e.g., position, force, velocity, etc., are updated. In a traditional explicit-time finite-element simulation, simulation time advances at a uniform rate for all mesh elements. In AVI, in contrast, elements advance in time with different time-steps.

In a sequential implementation, elements are updated in strict time order by maintaining a priority queue of elements and processing elements in time order. Figure 2 shows an outline of AVI; $t(w)$ is the time at which element w should execute. One way to parallelize AVI is to advance time in lock-step over the entire mesh; when the global clock is at time τ, all elements whose scheduled time or time-stamp are τ are processed in parallel before time is advanced (if some of these elements share vertices, independent subsets are processed in rounds). In this paper, we call this synchronous or *level-by-level* execution. This is a common parallelization technique for problems like breadth-first search in power-law graphs [26], but it is not useful for AVI, as we show in Figure 5, because few elements have the same time-stamp.

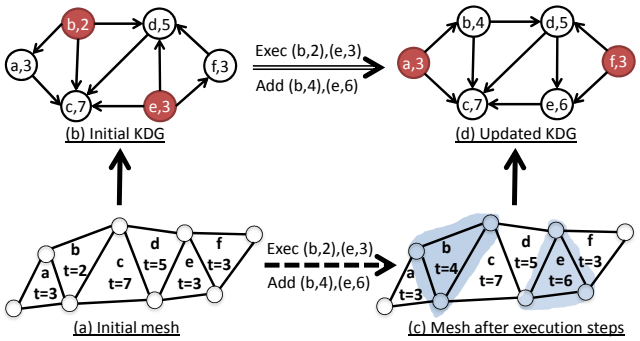


Figure 3: Using the KDG to parallelize AVI

```
1  AVI-BuildTaskGraph(Tasks W, TaskGraph G):
2    for (Task w ∈ W):
3      G.addNode(w)
4    for (Node w ∈ G):
5      for (Node w' ∈ G):
6        if (M.vertices(w) ∩ M.vertices(w') ≠ ∅):
7          if ((t(w) < t(w')) ∨ (t(w) = t(w') ∧ w ≺ w')):
8            G.addEdge(w → w')
9          else:
10           G.addEdge(w' → w)
11
12 AVI-ExecAndUpdate(Task wP, TaskGraph G):
13   //update state of corresponding element and
14   //discover new tasks
15   Task wC = Execute(wP)
16   if (t(wC) < endtime):
17     G.addNode(wC)
18     //assert M.vertices(wP) = M.vertices(wC)
19     for (Node n ∈ G.neighbors(wP)):
20       if (t(wC) < t(n) ∨ (t(wC) = t(n) ∧ wC ≺ n)):
21         G.addEdge(wC → n)
22       else:
23         G.addEdge(n → wC)
24   //as in conventional task graphs,
25   //remove node and incident edges
26   G.removeNode(wP)
27
28 AVI-SafeSourceTest: //any source is safe source
```

Figure 4: KDG Procedures for AVI

2.2 Asynchronous parallel execution of AVI

To exploit more parallelism, elements with different time-stamps must be processed concurrently, an approach that we call *asynchronous* parallel execution. One possible approach is to build a task graph and execute sources in parallel. Unfortunately, when tasks can create other tasks, it is not always safe to execute sources of the task graph in parallel.

As an example, consider Figure 3(b), which shows the task graph for AVI for the mesh in Figure 3(a). It has a node for the pending update to each element, and it has a dependence edge $(w_1 \rightarrow w_2)$ if the elements represented by tasks w_1 and w_2 have a vertex in common and the time-stamp of w_1 is less than the time-stamp of w_2. If they have the same time-stamp, as tasks $(e, 3)$ and $(f, 3)$ do, the tie is broken using some total order $\prec$ on tasks, such as the lexicographical order on element names. Procedure

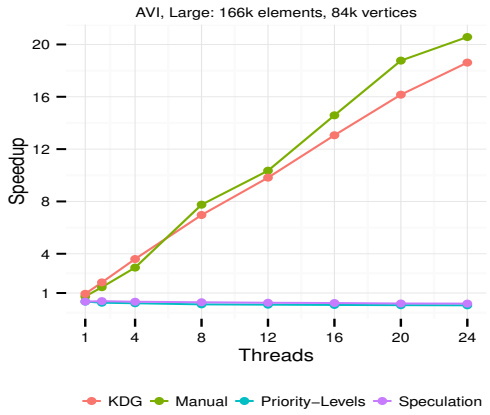


Figure 5: Speedup of different parallel implementations for AVI relative to an optimized serial implementation

`AVI-BuildTaskGraph` in Figure 4 shows pseudocode for building this graph.

There are two sources in the task graph, task $(b, 2)$ and task $(e, 3)$. It is safe to execute task $(b, 2)$ since it has the lowest time-stamp among all tasks in the system, but executing task $(e, 3)$ in parallel with $(b, 2)$ can result in an incorrect execution because, in principle, $(b, 2)$ could create a new task $(c, 2.5)$ that reads and writes the same vertex as task $(e, 3)$, and this new task should be executed before task $(e, 3)$ even though $(e, 3)$ is a source in Figure 3(b).

Although task graphs are not sufficient for this application, examination of the AVI code reveals that if a task w_P (parent task) is executed, it creates at most one new task w_C (child task), which must have a later time-stamp; also both w_P and w_C read and write the same portion of the mesh. The net effect in the task graph of executing w_P is to replace w_P with w_C and to flip the directions of some incident edges due to the new time-stamp. The effect of executing task w_P can be reflected in a simple, local update to the task graph as shown in procedure `AVI-ExecAndUpdate` in Figure 4. We also note that procedure `AVI-ExecAndUpdate` in Figure 4 never adds an edge to source in the original task graph. We call this the *stable-source property*, and it is useful because it means that a source in the task graph can always be safely executed, just like in conventional task graphs. In Figure 3(b) therefore, it is safe to execute tasks b and e in parallel; the task graph is incrementally updated whenever a task is completed, producing the task graph shown in Figure 3(d) after both tasks are completed.

To summarize, the pseudocode of Figure 4, together with the application code, is an implementation of the abstract diagram in Figure 1 for the particular case of AVI. The vertical arrows in Figure 1 are implemented by procedure `AVI-BuildTaskGraph`, the safe source test is implemented by `AVI-SafeSourceTest`, the horizontal arrows at the bottom of the diagram are implemented by the application code, and the horizontal arrows at the top of the diagram are implemented by procedure `AVI-ExecAndUpdate`, which invokes the application code.

2.3 Experimental results

Figure 5 shows experimental results that demonstrate the performance potential of using the KDG for parallelization. All experiments were performed on an Intel Xeon E7540 with 24 cores running at 2 GHz. To provide a sequential baseline, we implemented an AVI code using a priority queue to schedule tasks. To provide a reference parallel implementation, we implemented the AVI-specific edge-flipping DAG approach of Huang *et al.* [19]. In Figure 5, we call this implementation Manual. The line labeled KDG shows the performance our KDG-based runtime, described in more detail in Section 3; we see that this general and automatic parallelization approach achieves performance comparable to the application-specific approach of Huang *et al.*

In order to study existing parallelization schemes for ordered tasks, we implemented (i) a speculation-based executor, based on the work by Kulkarni et al. [25], which generalized the idea of out-of-order execution and in-order commit from TLS [36, 37], and (ii) a level-by-level executor. Figure 5 shows that thread-level speculation does not perform well for AVI. In most of our applications, including AVI, tasks take a very short amount of time, and most of the execution time is spent waiting on the commit queue, which limits scalability. More sophisticated commit mechanisms combined with hardware support may make speculation a viable parallelization strategy for our applications, but this remains to be seen. Figure 5 also shows the performance of the level-by-level executor (Priority-Levels) is poor due to little parallelism per level and large number of levels. For the input mesh used here, we observed that the level-by-level executor created 1.47×10^7 different levels, and the average number of elements per level was just 1.38.

3. Programming model and the KDG

Section 3.1 introduces the programming model. Section 3.3 describes the KDG formally. We describe two different approaches for parallelizing programs using the KDG. The first approach (§3.4) builds the KDG explicitly and updates it incrementally during execution. The second approach (§3.5) maintains the KDG implicitly by keeping sufficient metadata to find only sources in the KDG. These two approaches make different tradeoffs, as explained later. In our current system, these approaches are implemented by pieces of code called *executors*, which are concurrent schedulers based on different implementations of the KDG, and incorporate a few optimizations described in Section 3.6. Application programmers use flags in their code to choose an executor and the desired optimizations. We show the code for AVI in Section 3.7 to describe the programming interface.

3.1 Programming model

Unlike other task-based parallelization systems, such as OpenMP 4.0, XKaapi, and SMPSs, where the user invokes tasks, we provide *ordered foreach* loop to express paral-

lelism in an ordered algorithm. Ordered foreach is an ordered set-iterator [34], which iterates over a set of work items ordered by some priority metric. In the AVI pseudocode, shown in Figure 2, the *foreach-orderedby* statement shows an example of the ordered set iterator. The ordered foreach iterates over Q, to which new elemental updates are added inside the loop body. The *orderedby* clause of the ordered foreach specifies the priority ordering of tasks, which in the case of AVI is the timestamp of an elemental update. A *task* under this programming model is an iteration of ordered foreach loop. The loop body can contain code that reads or modifies shared data. The programmer must use concurrent data structures from Galois library [15, 34], which are thread-safe and contain hooks into our runtime so that the runtime can monitor accesses of a task (iteration) to shared data. The shared data structure in the case of AVI is the mesh M, and the values on its vertices are updated inside the loop body. Notice that this programming model is implicitly parallel: there are no threads, locks or barriers in the code. Also, serial code would look very similar, where instead of a an ordered foreach iterating over a worklist, the programmer would use a while loop iterating over a priority queue.

The execution model is similar to Galois [15, 34], where a main thread starts executing the serial portion of the code. Upon encountering the ordered foreach loop, the runtime will build a KDG internally, using the elements in the worklist and execute the safe sources from the KDG in parallel on a thread pool. An implicit barrier follows after the loop finishes and serial execution resumes from there.

Tasks under our programming model are iterations of the ordered foreach loop, which perform computation and read and write data locations such as the mesh in the AVI application. The set of global data locations read or written by a task is called the *rw-set* for that task (for AVI, the rw-set for a task (e, t) is the time-stamp of element e and its vertices). We require tasks to be *cautious*: a task must read all elements in its rw-set before it writes to any of them [34]. In our experience, this is not a significant restriction since many applications are naturally written like this while others can be transformed so that all the shared data is read before the first modification. Cautiousness is a useful property because the rw-set of a task can be determined by executing it up to the first global data structure write; since no updates are made to global data structures in this process, rw-sets can be computed without having to buffer or undo changes to global state.

The only requirement on the relative order in which tasks execute is that the final result should be identical to that obtained by executing the tasks atomically in the application-specific priority order, specified using the *orderedby* constraint on the iterator, such as the time-stamp order in the AVI code. This order is a total order, like time, in most applications, but it is useful to generalize it to a partial order to handle applications like tree traversals. We will assume that the order is specified by a function $t : W \rightarrow P$, where W is the domain of tasks for the application, and P is a partially ordered set $(T, \leq)$. One legal execution order is a sequential execution that always picks a task minimal in P from the worklist. Following standard notation, we will write $t(w_1) = t(w_2)$ if $t(w_1) \leq t(w_2) \wedge t(w_2) \leq t(w_1)$; this means that the priorities of the two tasks are the same. Relational operators such as $>$ and $\not>$ are defined similarly. If $t(w_1) < t(w_2)$, we will say that the priority of task w_1 is *earlier than* that of task w_2, and that the priority of task w_2 is *later than* that of task w_1.

We have implemented this programming model as a C++ template library. In Section 3.7, we describe its implementation in C++.

3.2 Program Properties

Program properties can be used to optimize the KDG implementation. In the following, we have identified some general properties found in many ordered programs. In §3.6, we describe how these properties can be used to optimize the KDG runtime. In §4, we discuss how sometimes problem-specific properties can be used to further customize the KDG implementation manually, in the user code.

An important class of algorithms are those for which every source in the KDG is a safe source.

DEFINITION 1. *An application is* **stable-source** *if every source in its KDG is a safe source.*

This property not only forms the basis of classical DAG-based scheduling but is also present in more complex applications, such as AVI, Kruskal's Minimum-weight Spanning Tree algorithm and Tree Traversals. Algorithms where not all sources are safe are called *unstable-source*. Examples are Discrete-Event Simulation (DES), and Asynchronous Collisions Simulation (Billiards).

Another important property is *monotonicity*. For example, in AVI, the time-stamp of a new task is always greater than the time-stamp of the parent task that created it. This property is defined below in a generalized form in which the new task may have the same priority as the parent task. Note that it applies even when priorities form a partial order.

DEFINITION 2. *An application is* **monotonic** *if whenever the execution of a task w_P creates task w_C, $t(w_P) \leq t(w_C)$.*

In conventional task graphs, the execution of a task does not change dependences between other existing tasks; i.e., the set of data locations that form the rw-set of a task is not a function of values updated by other tasks. We call this property *non-increasing rw-sets*.

DEFINITION 3. *An application has* **non-increasing rw-sets** *if the execution of a task does not introduce new locations in the rw-sets of other tasks.*

This property implies that dependences among existing tasks need not be recomputed during execution. Besides conventional task graphs, this property is observed the more complex applications considered in this paper such as AVI, DES and tree traversals. A counter-example of this property is Kruskal's MST algorithm, where rw-set of a task increases when components are merged. Notice that the property ignores the possibility of execution of a task causing locations to be removed from rw-set of other tasks. A stronger form of non-increasing rw-sets property called *structure-based rw-sets* is also seen in AVI, DES and traversals etc:

DEFINITION 4. *An application has* **structure-based rw-sets** *if the locations that comprise the rw-set of a task w_c are either (i) not a function of values updated by other tasks, or (ii) w_c is created by a parent task w_p and rw-set of w_c is a subset of w_p.*

This property allows the computation of the rw-set of a task correctly even while other tasks are being executed.

3.3 KDG

Given an application, let V be the domain of values, and let L be the set of abstract locations (for AVI, V might contain floating-point numbers and L might contain vertices, edges and elements for example). States are functions $\Sigma : L \rightarrow (V + L)^*$ (for example, in a minimal spanning tree (MST) computation, the abstract location for a weighted edge might contain the two end points of the edge and a value that is the edge weight). Tasks are functions from states to states $W : \Sigma \rightarrow \Sigma$. A task graph is an acyclic graph whose nodes correspond to some subset of W.

DEFINITION 5. *Given a domain of tasks W, a* task graph *is an acyclic directed graph $G = (V, E)$ where $V \subseteq W$, and $E \subseteq V \times V$.*

We let G_W denote the domain of task graphs for W.

DEFINITION 6. *Given a domain of tasks W, a* kinetic dependence graph *(KDG) is a three tuple $\langle G, P, U \rangle$ where*

- *$G : G_W$ is a task graph,*
- *$P : G_W \times \Sigma \times W \rightarrow \{true, false\}$ is a* safe source test*, and*
- *$U : G_W \times \Sigma \times W \rightarrow G_W \times \Sigma$ is an* update rule.

The safe source test P takes as input a task graph, the state of the computation, and a task, and returns true if the task is a safe source and can be executed in the current state of the computation. The update rule U takes as input a task graph, the state of the computation, and a task (which must be a safe source), and executes that task to update the task graph and the state. This update rule invokes the application code to execute task.

Not all triples $\langle G, P, U \rangle$ are semantically meaningful. KDGs of interest must satisfy two conditions called *safety* and *liveness*. Safety says that the safe source test and the update rule must be mutually consistent: applying the update rule repeatedly to the task graph never creates an incoming edge to a safe source. Liveness ensures forward progress: it says that if the task graph is non-empty, the safe source test must be true for at least one task of earliest priority. In the rest of the paper, we consider only KDGs that satisfy the safety and liveness properties.

- *Safety:* If w is a safe source in a KDG $\langle G, P, U \rangle$, the transitive closure of U applied to G never creates an incoming edge to w.
- *Liveness:* For any non-empty task graph G and state σ, the test $P(G, \sigma, w)$ must be true for at least one task w in G for which $t(w)$ is minimal among the tasks in G.

3.4 Explicit KDG Implementation

This section describes a general approach to parallelizing applications by maintaining a KDG explicitly. In the pseudo-code, we use **for** to indicate standard sequential loops, **doall** to indicate parallel loops with independent iterations, and **foreach** to indicate parallel loops whose iterations may have overlapping reads and writes. Any implementation of a foreach loop must guarantee that iterations are serializable. In the loops in this paper, we use fine-grain locking to implement concurrent data structure operations by hand. A more general technique would be to use the Galois system [34].

Building the KDG: Given the initial set of tasks, we can compute the rw-set of each task by executing it up to the point in that task where the first write to a global data location occurs, keeping track of all reads to data locations. Given the rw-sets and priorities of all tasks, one can build the task graph. As in the case of AVI, some total order on tasks, such as the lexicographical order on task names, is used to break ties between tasks of the same priority.

Incrementally updating the KDG requires keeping track of the rw-sets of all tasks. It is convenient to think of the association between tasks and their rw-sets as a bipartite graph B in which the two sets of nodes are tasks and data locations; if location l is in the rw-set of a task w, there is an undirected edge (w, l) in graph B. Thus, the rw-set of task w is simply the set of its neighbors in B.

Procedure `General-BuildTaskGraph`, shown in Figure 6, builds the task graph G and the rw-set graph B for a set of tasks. This is a generalization of procedure AVI-BuildTaskGraph. In line 8 of the procedure AddTask, the rw-set is computed by executing task w to its fail-safe point, which is in respect to the state seen by w before the fail-safe point denoted by σ_w in the figure. In line 10, the neighbors of l in B are the tasks whose rw-sets contain location l.

Executing a task and updating the KDG: In Figure 6, procedure `General-Execute` executes a task w_{P} by invoking the application code, which reads and updates some subset of state denoted by $\sigma_{w_{\mathrm{P}}}$. If new tasks are created, they are returned by the Execute method. Procedures `General-Execute` and `General-Update` then update the KDG. There are three kinds of updates that must be performed:

```
1  TaskGraph G
2  RWSets B
3  State σ
4
5  AddTask(Task w):
6    G.addNode(w)
7    B.addNode(w)
8    Locations RW = ComputeRWSet(w, σ_w)
9    for (Location l ∈ RW):
10     for (Task w_i ∈ B.neighbors(l)):
11       if ((t(w_i) < t(w))
12       ∨(t(w_i) ≠ t(w) ∧ w_i < w):
13         G.addEdge(w_i → w)
14       else:
15         G.addEdge(w → w_i)
16     B.addEdge(w ↔ l)
17
18 RemoveTask(Task w):
19   G.removeNode(w)
20   B.removeNode(w)
22 General-BuildTaskGraph(Tasks T):
23   foreach (Task w ∈ T)
24     AddTask(w)
25
26 General-Execute(Task w_P):
27   w_P.newTasks = Execute(w_P, σ_{w_P})
28   w_P.neighbors = G.neighbors(w_P)
29   //subrule R
30   RemoveTask(w_P)
31
32 General-Update(Task w_P):
33   //subrule N
34   for (Task w ∈ w_P.neighbors):
35     RemoveTask(w)
36     AddTask(w) //recomputes rw-sets
37   //subrule A
38   for (Task w_C ∈ w_P.newTasks):
39     AddTask(w_C)
40 KDG-RNA-Executor(Tasks initial):
41   General-BuildTaskGraph(initial)
42   while (G.notEmpty()):
43     Tasks E = {}
44
45     //parallel phase 1
46     doall (Task w ∈ G.sources()):
47       //apply safe source test
48       if (P(G, σ, w)):
49         E = E ∪ {w}
50
51     //parallel phase 2
52     foreach (Task w ∈ E):
53       General-Execute(w)
54
55     //parallel phase 3
56     foreach (Task w ∈ E):
57       General-Update(w)
```

Figure 6: General procedures for Explicit KDG (KDG-RNA): initial construction, update and execution

- *Subrule Remove:* Nodes corresponding to w in the task graph G and the rw-set graph B must be removed, together with incident edges.
- *Subrule Neighbors:* Updates to the global state by w may change the rw-sets of some pending tasks. These tasks share state with w, and must be immediate neighbors of w in the task graph G. The rw-sets of these tasks must be recomputed, and edges in G and B must be updated to reflect these new sets.
- *Subrule Add:* New nodes and edges must be added to G and B for each new task created by the execution of w.

We will refer to this basline explicit KDG implementation as *KDG-RNA*, based on the above three subrules.

Parallel execution: We combine the basic KDG procedures into a parallel executor. The executor proceeds in rounds; each round contains three phases, which are executed in parallel and are separated by global barriers.

In the first phase (Figure 6 lines 46–49), the executor finds safe sources by applying the safe source test in parallel. The safe source test reads the global state σ but does not write to it, so this phase is completely data-parallel.

Phases 2 and 3 apply the procedures `General-Execute` and `General-Update` over all the safe sources.

In general, executing a task (line 27 in Figure 6), must not be done concurrently with computing the rw-set of other tasks (line 8), which is reachable via lines 36 and 39. This is because executing a task may update state that is read while computing the rw-set of another task (i.e., σ_{w_P} in line 27 and σ_w in line 8 are not disjoint). Performing all tasks executions first and then updating the KDG second avoids this problem. Section 3.6 shows how to exploit certain program properties that allow these two phases to be done concurrently.

In phase 2, executing safe sources (Figure 6 line 27) is data-parallel because sources have disjoint rw-sets, but removing tasks (Figure 6 line 30) requires some synchronization because multiple sources may share neighbors in both graphs.

Phase 3, which implements subrules N and A, can also be done in parallel as long as adding and removing nodes and edges of the task graph and rw-sets are properly synchronized. As stated earlier, we use simple atomic operations and fine-grained locking to implement these graph manipulations.

3.5 Implicit KDG Implementation

The cost of explicit KDG, described in Section 3.4, is that it maintains the association between tasks and their rw-sets (through graphs G and B). In this section, we describe an *implicit KDG* (IKDG) executor that finds KDG sources without building the graph G, or the rw-sets data structure B explicitly, but at the cost of having to compute rw-sets of tasks repeatedly.

Parallel execution proceeds in rounds, where each round consists of three parallel phases separated by global barriers.

Phase I: The runtime executes each task up to its first global write, and tries to mark all the locations in its rw-set with the ID of the task. Marks are updated atomically using primitives like compare-and-set (CAS instruction). A task with earlier priority overwrites the marks made by another task, so that at the end of the phase, each location stores the ID of the earliest priority task that accessed it.

Phase II: Each task w checks if its marks have been changed since the first phase. If so, there is another task that has an overlapping rw-set and has precedence. Otherwise, task w has precedence over all tasks whose rw-sets overlap with its own; therefore, the task is a source. The safe-source test is applied to such tasks, and the task is added to a work-list if it passes this test.

Phase III: Tasks in the work-list of safe-sources are executed in parallel, and all marks are reset to an unused value.

3.6 KDG optimizations

The baseline KDG implementations described above can be expensive for applications with fine-grained tasks that perform relatively simple operations. Therefore, it is crucial

to identify opportunities to optimize the KDG implementation. Starting from the algorithmic properties specified by the user (introduced in §3.2), we can optimize the representation of the KDG and code required to maintain it. We can broadly group the optimizations in to following categories.

3.6.1 Windowing

Rather than compute the KDG for *all* tasks as is done in Figure 6, we can build the KDG for a subset of tasks, provided this subset forms a *prefix* of the set of tasks under priority ordering, i.e., if a task (w, t) with priority t is included in the subset, then all tasks with priority lesser than (or equal to) t are included in the subset. Pending tasks can be added to the window incrementally, as tasks in the window complete. We choose the size of the window adaptively: if threads lack work in the current window, the size of the next window is increased. Newly created tasks must be processed in a way that satisfies the prefix condition; for example, if t_{max} is the latest priority in the current window and a task of priority $t < t_{\text{max}}$ is created, it must be processed within the current window.

This optimization can be applied to both the explicit and implicit KDG. It controls the cost of graphs G and B for explicit KDG, and the cost of repeatedly marking rw-sets for implicit KDG.

The level-by-level executor described in Section 2.1 can be viewed as a particular windowing strategy for the implicit KDG in which windows correspond to priority levels. Implementations of level-by-level execution usually assume that all sources with earliest priority are safe sources, an assumption that is satisfied by monotonic algorithms (Definition 2) such as BFS and AVI. Level-by-level execution is useful when the number of priority levels is small, as is the case for BFS on social network graphs.

3.6.2 Update Rule Optimizations

No-Adds: The execution of a task does not create new tasks, so subrule A can be eliminated.
Non-increasing RW-sets: The execution of a task does not add new elements to the rw-sets of other tasks, so subrule N can be eliminated.
No-Adds + Non-increasing RW-sets: When an application has non-increasing rw-sets and no new tasks are created during execution, the KDG executor can be simplified to a conventional task graph scheduler (update rule is subrule-R).

3.6.3 Removal of parallel phases and barriers

Stable-source: If the application is stable-source, all sources are safe sources and no safe source test is needed. In these applications, phase 1 of the explicit KDG executor can be eliminated, and the loop beginning on line 52 of Figure 6 and the one beginning on line 56 can be rewritten to iterate over sources in G. Similarly, phases II and III of IKDG implementation can be combined into one phase, after eliminating the safe-source test.

Local Safe-source test: In general, a safe source test $P(G, \sigma, w)$ is a function of the entirety of σ. For some applications like discrete event simulation, P is, at most, only a function of σ_w (the state accessed by task w). In this case, phase 1 and phase 2 of the KDG-RNA executor can be combined.
Structure-based RW-set: When an application has structure-driven rw-sets, the barrier between phase 2 and phase 3 of the KDG-RNA executor can also be eliminated. This is because the portion of σ needed for computing the rw-sets (line 8 in Figure 6) is not affected by the execution of other tasks (line 27), and the intervening global barrier can be eliminated. When the application is stable-source, the optimized executor is asynchronous, i.e., does not need multiple rounds.
Comments: In our current system, programmers must request these optimized KDG versions through flags indicating algorithmic properties. Compiler analysis of the application code can determine some of these algorithmic properties. If no algorithmic properties are specified, our runtime falls back to IKDG executor with windowing optimization as the default.

3.7 Example: Galois program for AVI

```
1  Mesh M; // Galois data structure
2
3  // 1. work item
4  struct Update {
5    Element* elem; double t; // time stamp;
6  };
7  // 2. priority comparator
8  auto comparator = [&](const Update& u1, const Update& u2){
9    return u1.t < u2.t;
10 };
11 // 3. visit RW sets (prefix of loop body)
12 auto visitRWsets = [&] (Update& up) {
13   for (Vertex v: M.vertices(up.elem)
14     Runtime::write(v); // intent to write
15 };
16 // 4. loop body
17 auto applyUpdate = [&] (Update& up, W& wlHandle) {
18   Vec initial;
19   for (Vertex v: M.vertices(w.elem)):
20     initial.push(v.data);
21   Vec updated = AVI_update(up.elem, initial);
22   int i = 0;
23   for (Vertex v: M.vertices(w.elem)):
24     v.data = updated[i++];
25
26   up.elem->timestamp += up.elem->step;
27
28   if(up.elem->timestamp < endtime)
29     wlHandle.push(Update(up.elem, up.elem->timestamp));
30 };
31 // 5. initial set of work items
32 vector<Update> init;
33 for(Element e: M.elements())
34   init.push_back(Update(e, e->timestamp));
35
36 // 6. Ordered loop
37 Runtime::for_each_ordered(init.begin(), init.end(),
38   comparator, visitRWsets, applyUpdate,
39   Runtime::has_structure_based_rw_sets, // 7. flags
40   Runtime::is_stable_source,
41   Runtime::chunk_size<4>()); // and hints
```

Figure 7: AVI using KDG-based programming model

To make the programming model concrete, we present a portion of the Galois program for AVI in Figure 7. This C++ program corresponds to the pseudo-code for AVI shown

in Figure 2. Because the Galois system is library-based, there are differences between the pseudocode of Figure 2 and the C++ code, but the overall structure is similar; the main difference is the loop body in the pseudo-code must be split into two parts, a prefix that visits the rw-set (the lambda called `visitRWsets` in Figure 7) and a suffix that implements the rest of the loop body (the lambda called `applyUpdate` in Figure 7. The lambda for loop body takes two parameters, the item to work on and a handle to the worklist, which is maintained internally by the runtime. The mesh M is implemented using a graph implementation from the Galois library. The ordered foreach is a library call, similar to `std::foreach`, but takes extra parameters: (i) a priority-based comparison function, which implements the $orderedby$ clause, (ii) a set of initial tasks, (iii) flags to choose the executor and to enable optimizations, and (iv) for unstable-source algorithms, a safe source test.

4. Applications

In this section, we describe the applications in our benchmark suite. For each one, we describe (i) the program properties that we specified to our runtime in order to choose the best automatic KDG executor (user code written in our programming model §3.1), (ii) problem-specific optimizations to the KDG, done manually by implementing the KDG in the user code, and (iii) non-KDG parallel implementations.

4.1 AVI

AVI is described in Section 2.1.

Automatic executor: AVI is stable-source, and although it creates new tasks, new task's rw-set is the same as the parent tasks, therefore it has structure-based rw-sets. We choose the KDG-RNA executor, which gets optimized to an asynchronous executor that implements only subrule-R, and subrule-A.

Manual executor: This is a variation of the edge-flipping approach of Huang *et al.* [19]. We create an explicit dependence graph initially. Since (i) the rw-set of a child task is the same as its parent task (task that generated it), and (ii) the child task has a later time-stamp than the parent, we can update the node for the parent task in-place by changing the time-stamp and flipping its edges as needed.

Alternate parallelization: We obtained an MPI implementation from Adrian Lew [27] but it was orders of magnitude slower than our KDG implementation, so we did not use it.

4.2 Kruskal's MST algorithm

Kruskal's algorithm computes a minimal weight spanning tree (MST) of a weighted undirected graph [12]. It proceeds by applying edge contraction to the edges of a graph in increasing weight order. Each contraction is a task, and each edge successfully contracted is part of the MST. Figure 8 shows an example of edge contraction and its corresponding effect on the KDG. Tasks (nodes in KDG) are edges in the input graph. Edges that share a node in the input graph are neighbors in the task graph. Given two nodes a and b and the edge (a, b) between them, an *edge contraction* removes the edge (a, b) and replaces nodes a and b with a new node c that has the union of the edges incident on a and b. Edge contractions can create multiple edges between the same endpoints, in which case only the lightest weight edge should be retained. For efficiency, instead of explicitly removing an edge and performing the union of edges, edge contraction is implemented using a union-find data structure [12]. Another common optimization is to partition the edges into light and heavy edges. Light edges are processed first, and the result is used to filter edges connecting the same component amongst the heavy edges [6].

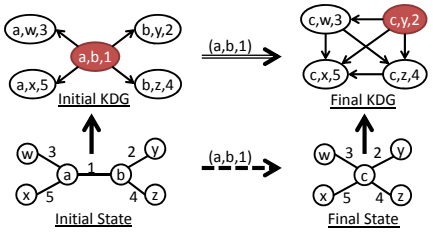


Figure 8: Using the KDG to parallelize Kruskal's algorithm

Edge-contraction may cause the rw-sets of existing tasks (edges in input graph) to increase, because the two nodes of edge being contracted are replaced by a new node with potentially larger adjacency set. For example, in Figure 8, edges incident on a and b are neighbors of each other in task graph, after c replaces (a, b). However, this algorithm is stable-source, since edge contraction only affects the neighbors of a source node in the task graph, which are not sources by definition.

Automatic executor: This application is stable-source but processing an edge may require recomputing the rw-sets of non-source tasks. We choose the IKDG executor with windowing.

Alternate parallelization: Blelloch et al. have parallelized this application by hand [6]. For capturing dependences correctly, they use *reservation stations* and priority-writes much in the same way as the IKDG exeuctor, but specialized to Kruskal, which does not generate new tasks and is stable-source.

Manual executor: Here we in-line the IKDG code into Kruskal application and specialize it by removing support for new tasks. The resulting implementation is similar to Blelloch et al., but with a different policy to control the window size.

4.3 Billiard Balls Simulation

Colliding billiard balls are a useful abstraction of complex phenomena such as molecular dynamics [2] and deformable contact [18]. The problem is stated as follows: given the positions and initial velocities of some balls on a billiards table, simulate the motion of these balls for some amount of time. Collisions are assumed to be elastic and to conserve both energy and momentum. Tasks are potential collision events. The rw-set of a collision consists of the two

balls involved in the collision. A collision is a source in the KDG if it is the earliest collision of both the balls involved. Simulating the collision between two balls a and b causes the rescheduling of events involving a and b with other balls. In the KDG, this requires adding new nodes for new collisions and marking some existing collisions as invalid.

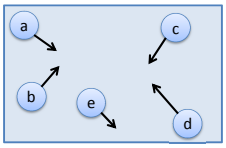


Figure 9: Billiards: Balls with initial velocities on a table

To see that this algorithm is not stable-source, consider the scenario shown in Figure 9 and suppose that collisions (a, b), and, (c, d) are sources in the KDG. Upon colliding with b, ball a may speed up and collide with c before c can collide with d, invalidating task (c, d). However, if we put a bound on how fast the balls can move, we can determine if the collision between balls c and d is sufficiently far away that even if a were to move with maximum velocity, it could not affect the collision between balls c and d. We use this *maximum velocity test* on all pairs of sources in the KDG as the safe-source test.

Automatic executor: We choose the IKDG executor over KDG-RNA because most of the non-source collisions become invalid.

Manual executor: Finding sources can be improved by keeping track of the earliest collision for each ball. The number of safe source test invocations can be reduced by maintaining per-thread priority queues to help determine earlier collisions.

4.4 Sparse LU factorization

We took the LU factorization code from Barcelona OpenMP Task Suite (BOTS) [14], which performs LU factorization (without pivoting) of a sparse matrix using *right-looking* algorithm [13]. It iterates left-to-right over the diagonal entries, of an $N \times N$ matrix M, and updates the bottom-right sub-matrix $M[i : N, i : N]$. The update consists of three parts: type I updates the diagonal entry at $M[i, i]$, type II updates the non-zero entries in the sub-row $M[i, j + 1 : N]$, and the sub-column $M[i + 1 : N, j]$, type III updates the sub-matrix $M[i + 1 : N, j + 1 : N]$ by computing $M[k, j] = M[k, j] - M[k, i] \times M[i, j]$, if $M[k, i]$ and $M[i, j]$ are non-zero; $M[k, j]$ may be zero before the update and may need to be allocated (i.e., *fill*) . Conceptually, we create a task of type I for each diagonal entry, a task of type II for each non-zero entry in the corresponding sub-row and sub-column, and a task for each entry of the sub-matrix that will be updated by type III update. Updates corresponding to diagonal entry i have priority over j if $i < j$, and, for a particular diagonal entry$i$: type I update must precede all type II updates, and all type II updates must precede any type III updates. In practice, to amortize the cost of task-creation and to improve locality, updates are performed on a block of the matrix at a time.

Alternate Implementation: The BOTS code implements a level-by-level parallelization approach, where tasks corresponding to one diagonal block are performed in two parallel phases. Type I update is performed serially, after which the first parallel phase performs updates of type II in parallel, while the second parallel phase performs updates of type III in parallel.

Manual Executor: We use the level-by-level parallelization similar to BOTS.

Automatic Executor: We modified the BOTS code to use the IKDG with windowing for parallelization. We perform a pre-processing pass similar to *symbolic factorization* that simply allocates blocks for the *fill* introduced by type III updates. The initial tasks are of type I, whose rw-set is all the non-zero entries in the sub-matrix $M[i : N, i : N]$ that are in the union of rw-sets of updates of type I, II and III corresponding to diagonal entry i. The rw-set of a type II task consists of the non-zero entry $M[i, k]$ in the row i that it updates, and the entries in the sub-column $M[i + 1 : N, k]$ that depend on $M[i, k]$ and will be updated by type III updates. This strategy enforces the precedence constraints among type I, II, III tasks for a particular diagonal entry i, and those among different diagonal entries. Additionally, the algorithm becomes stable-source and has structure-based rw-sets. We can also choose asynchronous KDG-RNA with subrules R and A, as in the case of AVI.

4.5 Discrete-Event Simulation

In discrete-event simulation (DES), there is a network of processing stations each of which has internal state. Stations communicate by sending events along FIFO links. When a station consumes an event, it may update its internal state and send events to some of its outgoing neighbors. In the simulation, events are assigned time-stamps and are processed in global time-stamp order to maintain time causality.

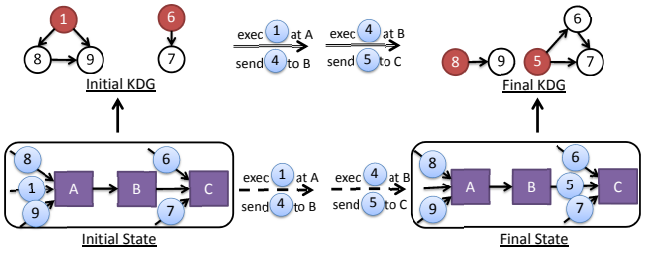


Figure 10: DES: Network of Stations A, B & C

A task corresponds to the processing of an event. The rw-set of an event is its target station. Hence the KDG has a node for every event, and edges among pending events on a particular station. DES is unstable-source, because a station can receive an event, on a particular input edge, that has a time-stamp earlier than events previously received on other inputs. For example, in Figure 10, events 1 and 6 are sources initially (we will refer to events by their time-stamps for

simplicity). Let us assume that station A processes event 1 and sends an event 4 to B. B processes event 4 and sends an event 5 to C. Now 5 is a source in the KDG, while 6 is no longer a source.

Chandy and Misra [9] observed that with FIFO-links, a station must receive events on a particular input edge in increasing time-order. Therefore, a station that has received at least one event on all inputs can safely process the earliest of them, regardless of the global time-stamp order. We devised a safe-source test for DES using this insight. The test is local and relatively cheap, but may fail to identify certain safe sources in a cyclic network.

Automatic executor: DES has structure-based rw-sets, and although it is not stable-source, our safe source test depends on local state only. The resulting KDG-RNA executor is asynchronous, just like AVI.

Alternate implementation: The Chandy-Misra approach sends null messages carrying timing information, and these messages are used by stations to determine when they can fire safely. We compare against Chandy-Misra implemented in Lonestar suite [15].

Manual executor: In DES the size of the rw-set of an event is exactly one (i.e., the receiving station). Therefore, the KDG can be represented by using a priority queue for each station, whose earliest event is a source in the KDG.

4.6 Breadth-first Search

A BFS task is a tuple (n, L), which represents updating the distance label (i.e., number of hops from some starting node) of node n to value L. These updates must be processed in increasing distance order. BFS is not stable-source. A common parallelization strategy for BFS is to process updates in a level-by-level fashion. We use this insight to devise a safe source test for BFS. A source task is safe if it has a distance label equal to the current global minimum level.

Automatic executor: For most inputs, the number of priority levels is small. We use the IKDG executor with level-by-level windowing strategy.

Manual executor: In BFS, only two levels need to be maintained at a time.

Alternate Implementation: We use parallel BFS code released by Leiserson et al. [26].

4.7 Bottom-up tree traversal algorithms

Bottom-up tree traversal requires visiting children of a node in tree before the node itself. In our experiments, we use a particular instance of bottom-up tree traversal: the center-of-mass computation in Barnes-Hut [3].

Automatic executor: This application is stable-source, has non-increasing rw-sets and does not create new tasks. We use the KDG-RNA executor with only subrule-R. Additionally, since the dependences are only between parent and child tasks, we disable the computation of rw-sets.

Manual executor: We embed the information for task graph in the tree being traversed because they are isomorphic. This is done when building the tree.

Alternate implementation: We used Cilk to parallelize the recursive implementation.

5. Evaluation

We performed all our experiments on an Intel Xeon E7-4860 server with 40 cores running at 2.2 GHz, organized in 4 packages with 10 cores each, and 24 MB L3 per package. The machine has 128GB of RAM, and runs Scientific Linux 6.3. All programs were compiled with GCC 4.8 at the -O3 optimization level; the only exceptions were Cilk-Plus implementations used for comparisons, which were compiled with the Intel Compiler 14.0. Figure 11a lists the small and large inputs used for each of our applications.

5.1 KDG Performance Comparison

To evaluate the relative performance of KDG-based implementations, we used two kinds of alternative implementations: (i) sequential implementations, to provide a baseline for speedup estimates, and (ii) third-party parallel implementations, when they were available (described in §4, and referred to as *Other* in Figure 11b). Sequential implementations do not use any parallel constructs. For LU, BFS, and MST, we use highly optimized serial implementations released by BOTS [14], Schardl [26], and Blelloch et al. [6], respectively. AVI, Billiards and DES use priority-queues; Tree Traversal is a recursive implementation.

We show results for two kinds of KDG-based implementations, as discussed in §4: (i) KDG-Auto: code written in our programming model that chooses an executor from our KDG-based runtime by specifying flags indicating program properties, (ii) KDG-Manual: KDG implemented in user code, customized to the application.

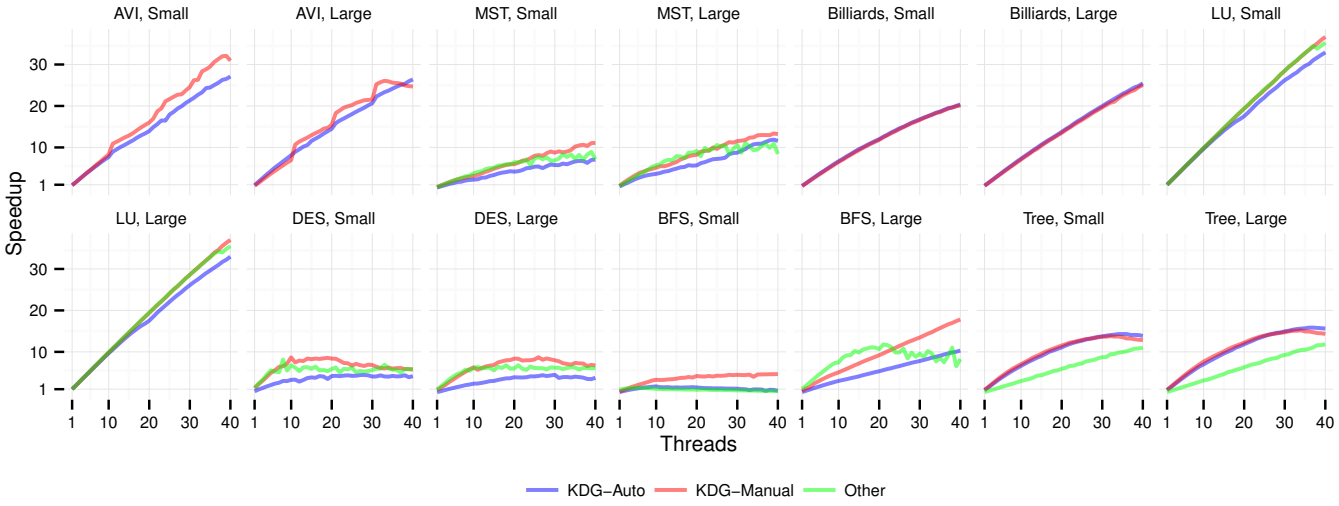
Figure 11b plots the speedup of KDG-Auto, KDG-Manual and Other against sequential baseline. The best running times of both sequential and parallel implementations are listed in Figure 11a. Both KDG-Auto and KDG-Manual achieve good parallel speedup for many applications, e.g., for AVI, KDG-Auto achieves a speedup of 27 on 40 cores, while for LU, the speedup is 33. Overall, KDG-Auto performs either better than, or within 10-20% of third-party parallel implementations, which often incorporate application specific optimizations. KDG-Manual performs equal to, or, better than third-party parallel implementations, which shows the true potential of the KDG abstraction. The performance difference between KDG-Auto and KDG-Manual are due to overheads related to KDG-Auto being general. Finally, for applications like AVI and Billiards, which are hard to parallelize by hand and no existing system supports their easy parallelization, KDG provides good parallel performance (best case speedups of 27 and 25 respectively).

5.2 Overhead of KDG executors

We analyzed the over-heads and scalability problems of automatic KDG runtime (KDG-Auto), using Vtune Amplifier to collect a profile of total cycles. Figure 12 shows a breakdown of total cycles aggregated over all the threads,

	Inputs		Serial		KDG-Auto		KDG-Manual		Other	
	Small	Large	Small	Large	Small	Large	Small	Large	Small	Large
AVI	42 K elements	166 K elements	55.5	98.1	2.0	3.7	1.7	3.7	-	-
MST	2D Grid $\|V\|$ = 16 M, $\|E\|$ = 33 M	Random $\|V\|$ = 67 M, $\|E\|$ = 268 M	10.0	82.1	1.4	6.9	0.9	6.1	1.1	7.4
Bill.	7 K balls, $7K \times 7K$ table	15 K balls, $15K \times 15K$ table	36.7	156.9	1.8	6.1	1.8	6.2	-	-
LU	$12K \times 12K$, 140 K non-zeros	$23K \times 23K$, 1.1M non-zeros	195.7	355.1	5.9	10.8	5.3	9.6	5.5	10.0
DES	12-bit Tree Multiplier	64-bit Kogge-Stone Adder	26.9	32.6	6.0	7.2	3.1	3.7	3.3	4.8
BFS	USA road $\|V\|$ = 23 M, $\|E\|$ = 59 M	Random $\|V\|$ = 67 M, $\|E\|$ = 268 M	1.3	18.2	0.78	1.8	0.28	1.0	0.98	1.5
Tree	40 M bodies, Plummer dist. [35]	100 M bodies, Plummer dist. [35]	10.1	28.2	0.70	1.77	0.73	1.85	0.92	2.4

(a) Inputs and best running times (in seconds)



(b) Speedup relative to best serial time, showing small input only (due to space limitations)

Figure 11: KDG performance

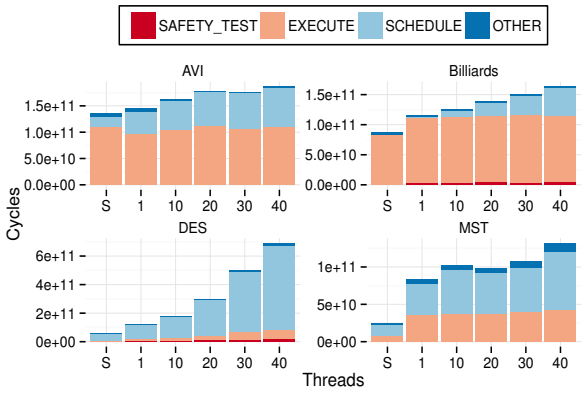


Figure 12: Breakdown of total cycles (summed over all threads). Small Input was used.

for a particular run. We chose four benchmarks for this study, using small inputs. Out of these, AVI and DES were configured to choose explicit KDG implementation from the runtime, while Billiards and MST were configured to choose implicit KDG implementation. The first bar, labeled *S*, shows the breakdown of serial baseline. We group the cycles from the profile into (1) SAFETY_TEST: safe source test (DES and Billiards only) (2) SCHEDULE: KDG maintenance (3) EXECUTE: task execution (4) Other: cost that could not be categorized, e.g., cycles spent in functions called by both user code and runtime. SCHEDULE for serial implementation accounts for the cost of maintaining priority queues or sorting etc.

In general, the cost of KDG (SCHEDULE) scales with the number of threads, with the exception of DES. Further investigation showed that DES suffers from low parallelism (small number of sources), which causes contention on the internal worklist for sources. DES has tasks that perform very little work, which magnifies the overheads of automatic KDG runtime. Notice that the bandwidth consumed by such contention causes EXECUTE to consume more cycles on higher number of threads in some cases. Additionally, task execution when using KDG executors takes longer than when using serial priority-queue because of the cache space and memory bandwidth consumed by KDG runtime.

5.3 Speculation and Level-by-Level Parallelization

In the following, we show experimental results highlighting the problems with using speculation and level-by-level parallelization for these applications (introduced in §2.3).

Figure 13 shows a breakdown of time spent by threads, when using the speculation-based executor, for three appli-

cations. We observed that the commit queue and sequential commits inhibit the scalability of the speculative executor. Tasks in our applications are fine-grain so threads spend most of their time waiting to commit. Additionally, the overhead of copying state and storing undo actions is significant.

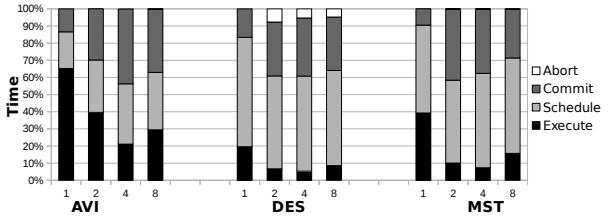


Figure 13: Speculative overheads:Breakdown of time spent by each thread

Application	#levels	avg. #tasks/level
AVI	1.47×10^7	1.38
BFS (on Random graph)	22	6.35×10^6
BFS (on Road Net.)	6262	4607
Billiards	6.03×10^5	1.12
DES	5.2×10^4	1717.21
MST	10^4	2.7×10^4
Tree Traversal	23	2.51×10^6

Figure 14: Amount of parallelism exposed by Level-by-Level executor. Number of levels indicates critical path; Average number of tasks per level indicates parallelism.

The performance of level-by-level execution depends on (1) the number of tasks within each level, which is a measure of parallelism, and (2) the total number of priority levels, which is a measure of the critical path. Figure 14 shows these statistics for our applications, using the level-by-level executor. The level-by-level executor performs well only for BFS (Large input) and Tree Traversal, where there are only a few levels and each level has lots of tasks. On the other hand, level-by-level execution does not discover any useful parallelism in Billiards and AVI. Timestamps are real numbers in these applications and two tasks rarely have the same timestamp. For DES, events have integer timestamps, while in MST, edge weights are integers. Therefore, level-by-level finds some parallelism, but it does not scale beyond a few threads. This is due to the overheads introduced by stopping threads at a barrier after completing each level.

6. Related work

The term Kinetic Dependence Graph was inspired by Guibas's Kinetic Data Structures [4], which are data structures used in computational geometry and graphics. These data structures capture time-varying information and allow incremental and efficient updates at runtime.

General parallelization approaches: Thread-level speculation (TLS) [24, 29, 36] is a general technique for exploiting parallelism in DO loops with statically unpredictable dependences. In our experience, speculation is useful for parallelization of atomic tasks that can commit in any order such as Galois-style unordered set iterators [34], on the other hand, the overheads of in-order commit make it less attractive for parallelizing tasks that must commit in a particular order.

Task graphs have been used for parallelization in dense linear algebra [38, 39], sparse linear algebra [17], and linear transforms [40]. The KDG is a generalization of classic task graphs because it can handle the creation of new tasks and changes in dependences during execution.

Fork-join style programming models such as Cilk [7], NESL [5] and X10 [10] require dependences to be between child and parent tasks only. Task graphs in these systems can be seen as a special case of the KDG. The restriction makes it difficult to express applications such as billiard balls, DES and AVI.

Application-specific parallelization: In the literature, there are many application-specific parallelizations of the benchmarks considered in this paper. However, most of them can be viewed as using application-specific specializations of the KDG.

For example, parallelization of AVI by Huang et al. [19] uses a dependence DAG with a specific update rule (*i.e.*, edge flipping) to incorporate newly created tasks. This is a special case of the KDG. The parallel implementation of Kruskal's MST algorithm in the PBBS benchmark suite [6] uses a special case of IKDG, optimized for the fact that the application has the stable-source property and does not generate new tasks. Our general IKDG executor was inspired by this work. Lubachevsky describes a billiard balls algorithm that is based on dividing up the physical region between processors, and processing events in different partitions in parallel. It exploits an upper bound on the velocity of balls to process in parallel balls that are sufficiently far away [28].

Discrete-event simulation has been parallelized using both optimistic techniques like Time Warp [21] and conservative techniques like the Chandy-Misra-Bryant algorithm [30]. Our safe-source test is based on Chandy-Misra's null messages, which allows stations to find safe events locally. Time Warp uses speculative execution, so it is quite different from the KDG-based approach described in this paper.

7. Conclusion

In this paper, we have described a new approach to parallelizing many algorithms with priority ordered tasks, dynamic task creation, and changing dependences. Our approach is based on the Kinetic Dependence Graph, which can be viewed as a generalization of the static dependence graph. The KDG is used to schedule tasks for parallel execution, and it can be computed and updated efficiently at runtime. We presented experimental results for several important ordered algorithms showing that our approach outperforms hand-tuned third-party parallel implementations in several cases and gives good parallel performance for other algorithms that are otherwise difficult to parallelize.

References

[1] S. Ainsley, E. Vouga, E. Grinspun, and R. Tamstorf. Speculative Parallel Asynchronous Contact Mechanics. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 31(6):8, Nov 2012.

[2] B. J. Alder and T. E. Wainwright. Studies in molecular dynamics. i. general method. *J. Chem. Phys. 31 (2)*, page 459, 1959.

[3] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(4), December 1986.

[4] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In *JOURNAL OF ALGORITHMS*, pages 747–756, 1997.

[5] G. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), March 1996.

[6] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 181–192, New York, NY, USA, 2012. ACM.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[8] M. G. Burk, K. Knobe, R. Newton, and V. Sarkar. The concurrent collections programming model. Technical Report Department of Computer Science Technical Report TR 10-12, Rice University, December, 2010.

[9] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Software Eng.*, 5(5):440–452, 1979.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[11] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 13–24, Vancouver, Canada, June 2000.

[12] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, editors. *Introduction to Algorithms*. MIT Press, 2001.

[13] T. A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[14] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[15] `http://iss.ices.utexas.edu/?p=projects/galois`.

[16] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. *Parallel and Distributed Processing Symposium, International*, 0:1299–1308, 2013.

[17] A. Gupta, S. Koric, and T. George. Sparse matrix factorization on massively parallel computers. In *SC '09*, 2009.

[18] D. Harmon, E. Vouga, B. Smith, R. Tamstorf, and E. Grinspun. Asynchronous contact mechanics. In *ACM SIGGRAPH 2009 papers*, SIGGRAPH '09, pages 87:1–87:12, New York, NY, USA, 2009. ACM.

[19] J.-C. Huang, X. Jiao, R. M. Fujimoto, and H. Zha. Dag-guided parallel asynchronous variational integrators with super-elements. In *Proceedings of the 2007 summer computer simulation conference*, SCSC, pages 691–697, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[20] Intel Corporation. Intel thread building blocks 2.0. `http://osstbb.intel.com`.

[21] D. R. Jefferson. Virtual time. *ACM TOPLAS*, 7(3), 1985.

[22] F. John. *Partial Diffeential Equations*. Springer Publishers, 1984.

[23] K. Kennedy and J. Allen, editors. *Optimizing compilers for modren architectures:a dependence-based approach*. Morgan Kaufmann, 2001.

[24] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, 48(9):866–880, Sept. 1999.

[25] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not. (Proceedings of PLDI)*, 42(6):211–222, 2007.

[26] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, 2010.

[27] A. Lew, J. E. Marsden, M. Ortiz, and M. West. Asynchronous variational integrators. *ARCHIVE FOR RATIONAL MECHANICS AND ANALYSIS*, 2003.

[28] B. Lubachevsky. Simulating colliding rigid disks in parallel using bounded lag without time warp. In *SCS Multiconference*, 1990.

[29] P. Marcuello, A. González, and J. Tubella. Speculative multithreaded processors. In *Proceedings of the 12th international conference on Supercomputing*, ICS '98, pages 77–84, New York, NY, USA, 1998. ACM.

[30] J. Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.

[31] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[32] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 4.0, July, 2013.

[33] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151, Sept 2008.

[34] 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, and X. Sui. The Tao of Parallelism in Algorithms. In *PLDI 2011*, pages 12–25, New York, NY, USA, 2011. ACM.

[35] H. Plummer. On the problem of distribution in globular star clusters. *Mon. Not. R. Astron. Soc.*, 71(460), 1911.

[36] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.*, 10(2):160–180, 1999.

[37] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA '00*, 2000.

[38] `http://icl.cs.utk.edu/dague/`.

[39] `http://www.cs.utexas.edu/~flame/web/`.

[40] `http://www.spiral.net/`.