6.886 Final Project: Bigger and Faster Data-graph Computations for Physical Simulations

Predrag Gruevski, William Hasenplaugh, and James J. Thomas

Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{predrag, whasenpl, jjthomas}@mit.edu
http://toc.csail.mit.edu/

Abstract. We investigate the problem of implementing the physical simulations specified in the domain-specific language Simit as a ***data-graph computation***. Data-graph computations consist of a graph $G = (V, E)$, where each vertex has data associated with it, and an update function which is applied to each vertex, taking as inputs the neighboring vertices. PRISM is a framework for executing data-graph computations in shared memory using a scheduling technique called *chromatic scheduling*, where a coloring of the input graph is used to parcel out batches of independent work, sets of vertices with a common color, while preserving determinism. An alternative scheduling approach is *priority-dag scheduling* where a priority function ρ mapping each vertex $v \in V$ to a real number is used to orient the edges from low to high priority and and thus generate a dag. We propose to extend PRISM in two primary ways. First, we will extend it to use distributed memory to enable problem sizes many orders of magnitude larger than the current implementation using a graph partitioning approach which minimizes the number of edges that cross distributed memory nodes. Second, we will replace the chromatic scheduler in PRISM with a priority-dag scheduler and a priority function which generates a cache-efficient traversal of the vertices when the input graph is locally connected and embeddable in a low-dimensional space. This subset of graphs is important for the physical simulations generated by the language Simit.

1 Introduction

The age of big data is upon us as organizations large and small are coping with massive volumes of data from sensors, website clicks, e-commerce, and more. One such type of big data problem is in the space of physical simulations (e.g. fluid dynamics, the n-body problem, computer graphics etc.). This paper investigates the specific problem of performing physical simulations expressed in the language Simit on very large datasets quickly and deterministically. Simit generates a static graph, as in Figure 1, which is typically locally connected and embeddable in a ND space, where typically $N = 3$.[1] Then, a function that operates on each vertex and its neighbors is applied to

[1] We use the notation ND to refer to an N-dimensional space.

all vertices over many (e.g. at least millions) time steps. These functions are typically some approximation of physical forces (e.g. Newton's laws). Our goal is to develop a software platform that performs these physical simulations in a fast and scalable manner on a distributed system of multi-core nodes.

In recent years, there has been growing interest in developing frameworks for the storage and analysis of this data on large compute clusters, Hadoop [3,4] being among among the most popular of these. Hadoop breaks up large datasets into pieces distributed across many shared-memory multi-core nodes in a cluster, each of which communicates via a message-based network protocol. Users supply computations, or ***map*** operators, that are evaluated over each of the pieces independently and other computations, or ***reduce*** operators, that combine the results. Many problems can be cast into the Hadoop model, but in many cases the Hadoop approach is far less effecient than more specialized methods optimized for graph algorithms, as we will explore throughout this paper.

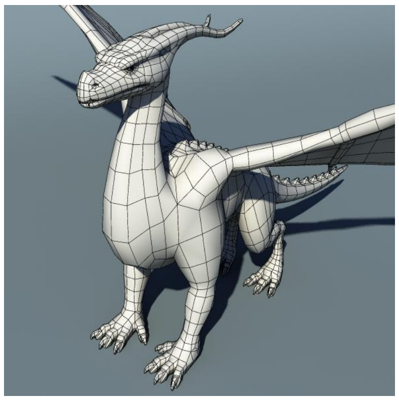
Fig. 1: A mesh graph where lines correspond to edges and intersections of lines correspond to vertices.

The idea behind recent big data frameworks, including Hadoop, is to decouple scheduling and data layout from the expression of the computation, enabling high programmer productivity and portable, best-in-class performance. However, iterative graph algorithms are one class of problems that is not well-suited to the Hadoop approach. In particular, each Hadoop computation writes its output to disk, so each iteration in iterative computations incurs the overhead of a disk write and then a subsequent disk read for the next iteration. In addition, graphs are difficult to split into completely independent sets (with no crossing edges) for the map phase of a Hadoop computation, so the maps are often wasteful. However, the idea of decoupling data and scheduling from the expression of the algorithm is very useful for designing frameworks for graph algorithms, even if Hadoop itself is ill-suited to the task.

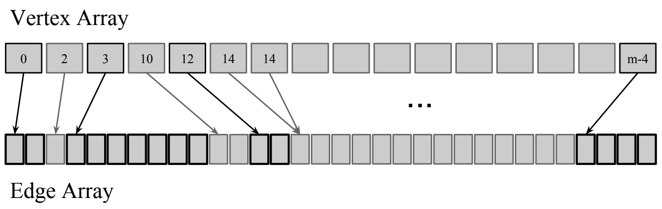

Fig. 2: Graphs are stored in memory on a single cache-coherent multi-core in a sparse-matrix format. The vertex array contains vertex data and an index into an edge array, which contains vertex IDs of the associated neighbors.

1.1 Data-graph Computations

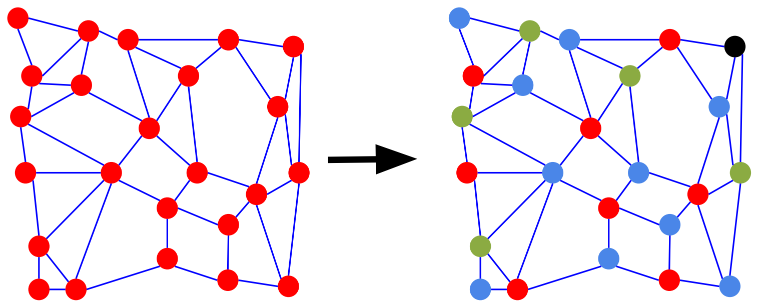

Fig. 3: Example of how a graph can be partitioned into independent sets of vertices denoted by color, each set of which is able to be executed simultaneously without causing data races. Iterating through the colors serially and executing the corresponding independent sets in parallel is a technique called *chromatic scheduling*.

In response to the shortcomings of applying Hadoop and similar systems to graph problems, Guestrin et al developed the GraphLab framework [13] for iterative graph algorithms, largely consisting of machine learning algorithms. In particular, GraphLab is a framework for implementing a ***data-graph computation***, which consists of a graph $G = (V, E)$, where each vertex has associated user-specified data and a user-specified ***update function*** which is applied to every vertex, taking as inputs the data associated with the associated neighbors. On each *round* or *time step* the update function is applied to all vertices. Many interesting big data algorithms, including Google's PageRank, can be easily expressed under this model. GraphLab comes in two variants – a multi-core implementation that attains parallelism by updating nodes concurrently on multiple processing cores, and a distributed implementation that additionally spreads vertices across multi-core nodes.

Faster, Determinstic Data-graph Computations

Recently, Kaler et al. [10] demonstrated that general data-graph computations could be made to be deterministic without giving up high-performance, in fact, while increasing performance. Their system PRISM is 1.2-2.1 times faster than the nondeterministic GraphLab framework on a representative suite of data-graph computations in a multi-core setting. The technique Kaler et al. proposed is called ***chromatic scheduling***.

In chromatic scheduling one finds a valid coloring of the graph as depicted in Figure 3, an assignment of colors to vertices such that no two neighboring vertices share the same color, and then serializes through the colors. Since each subset of the graph of a given color is an independent set (i.e. no two members share an edge) they may be processed simultaneously without causing a data race. This assumes that the update function applied to a vertex v reads the data associated with all of its neighbors $v.adj = \{w \in V | \langle v, w \rangle \in E\}$ and writes only the data associated with v. Chromatic scheduling is a powerful technique because it allows the parallel execution of a data-graph computation without any concurrent operations on data. This removes the overhead of mutual-exclusion locks incurred by GraphLab or other atomic operations that would be required in a design with concurrency.

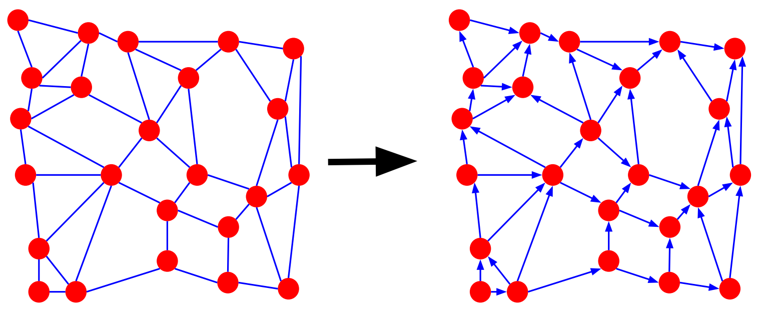

Fig. 4: An alternative to chromatic scheduling, which yields a deterministic, data race-free output, is dag scheduling. A priority funcrtion $\rho : V \to \mathbb{R}$ is used to create a partial order on the vertices, orienting an edge from low to high priority results in a dag. The vertices are processed in dag order: a vertex is not processed until all of its predecessors have been processed.

While chromatic scheduling does a good job of enabling high parallelism without any concurrency, it can be inefficient for cache usage. In Figure 2 we see the standard sparse-matrix representation used in PRISM and GraphLab. We can see that to process the update function of a vertex v of color c the worker needs to read data associated with all of its neighbors $v.adj$, but by virtue of being in different color sets by defintion, each vertex $w \in v.adj$ can not be processed until after all vertices of c have been processed. This potentially squanders the potential cache advantage of processing the neighbors of v soon after v is processed itself, while they are still in cache.

```
JONESPLASSMANN($G$)
1  let $G = (V, E, \rho)$
2  **parallel for** $v \in V$
3      $v.pred = \{w \in v.adj : \rho(w) < \rho(v)\}$
4      $v.succ = \{w \in v.adj : \rho(w) > \rho(v)\}$
5      $v.counter = |v.pred|$
6  **parallel for** $v \in V$
7      **if** $v.pred$ == $\emptyset$
8          JP-UPDATE($v$)

JP-UPDATE($v$)
9   UPDATE($v$)
10  **parallel for** $u \in v.succ$
11      **if** JOIN($u.counter$) == 0
12          JP-UPDATE($u$)
```

Fig. 5: The Jones-Plassmann parallel priority-dag scheduling algorithm. JONESPLASSMANN uses a recursive helper function JP-UPDATE to process a vertex using the user-supplied UPDATE function once all of its predecessors have been updated, recursively calling JP-UPDATE on any successors who are eligible to be updated. The function JOIN decrements its argument and returns the post-decrement value.

An alternative approach to chromatic scheduling is ***dag scheduling*** [9], depicted in Figure 4 and used extensively by Hasenplaugh et al. [7] in the context of graph coloring. In dag scheduling, the graph is turned into a dag through the use of a priority function $\rho : V \to \mathbb{R}$. In particular, an undirected edge connecting vertices v and w is oriented as

$\langle v, w \rangle$ if $\rho(v) < \rho(w)$ (ties are broken by comparing the vertex numbers). The vertices are then processed in dag order, meaning that a vertex v may be processed only once all of its predecessors $v.pred = \{w \in v.adj : \rho(w) < \rho(v)\}$ have been processed. A relatively simple implementation of dag scheduling JONESPLASSMANN described in Figure 5 involves initializing a counter at each vertex with the number of predecessors in the dag. Then, after a vertex v is updated the worker atomically decrements the counters for all successors $v.succ = v.adj \setminus v.pred$, recursively updating any successors whose counters drop to zero. This scheduling approach affords us the opportunity to process vertices shortly after they are read by their neighbors, a potential caching advantage. We will explore a technique for achieving such cache behavior for a special class of graphs corresponding to physical simulations in Section 3.

1.2 Simit

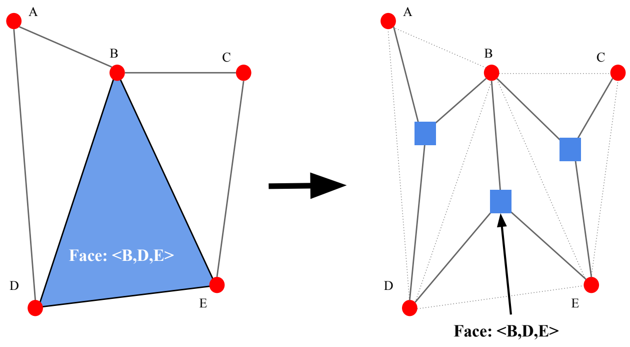

Fig. 6: Graphs generated by the language Simit feature hyperedges, an example of which is in blue on the left. Hyperedges are represented by different *types* of vertices in the resulting data-graph computation. The square vertices in the figure represent hyperedges and have associated per-hyperedge data.

Simit is a language used to describe physical simulations (e.g. fluid dynamics, the n-body problem, computer graphics etc.). Typically, Simit generates a mesh graph (i.e. a wire mesh discretization of a continuous 3D object) of an object in physical 3D space, like the one depicted in Figure 1. These meshes contain vertices at intersections of line segments, hyperedges (e.g. triangular faces) and tetrahedron volumes, each of which of these three constructs has associated data. An immediate hurdle presents itself when trying to cast operations on such a mesh graph as a data-graph computation: data-graphs do not natively support hyperedges or tetrahedra. However, Simit is a language and thus the compiler can intervene to represent the mesh graph as a data-graph where each vertex has a *type*.

We see in Figure 6 how a face (or hyperedge) connecting vertices B, D, and E, for example, can be represented as a new type of vertex (i.e. the blue squares in the figure) which is conntected to B, D, and E by individual edges. In addition, a tetrahedron can be viewed as a set of four adjacent triangular faces (or hyperedges). In Figure 7 we see such an example, where a third type of vertex (i.e. the purple diamond in the figure) is

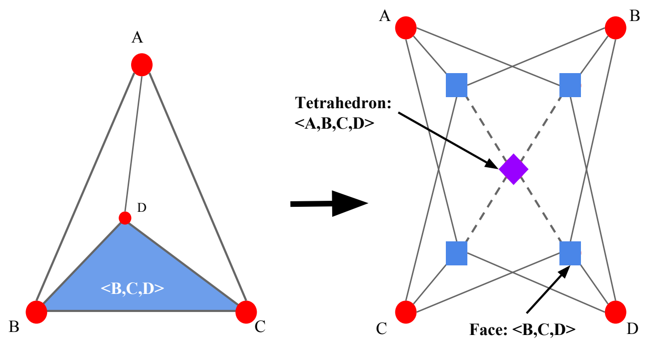

Fig. 7: Graphs generated by the language Simit have tetrahedrons, as depicted on the left above. A tetrahedron is composed of four hyperedges (or *faces*), an example of which is in blue on the left. Tetrahedra are represented by different *types* of vertices in the resulting data-graph computation. The diamond vertex on the right represents a tetrahedron and is connected to its four constituent hyperedges.

connected to four face type vertices. Finally, the update function, which is generated by the Simit compiler can generate an update function which takes the type of the vertex as a parameter and jumps to the relevant code as in a case statement.

1.3 The Hilbert Space-filling Curve

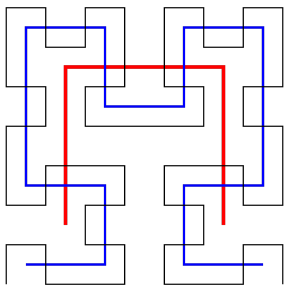

Fig. 8: Three recursion levels of a 2D Hilbert space-filling curve [8].

In this paper we propose a new priority function for use with dag scheduling of data-graph computations generated by Simit. This priority function can also be used to reorder the vertices in the graph to exploit improved cache behavior as we explore in Section 2. In particular, we use the bounding box of the graph in 3D space to normalize the graph to the unit cube. Then, we decompose the unit cube into a regular grid $2^k \times 2^k \times 2^k$ each grid point of which is assigned a scalar value by the Hilbert space-filling curve [8]. A 2D example of the Hilbert space-filling curve is given in Figure 8. The red curve is the first recursion level and illustrates the basic inverted 'U' shape. The blue curve shows how each quadrant is partitioned into four independent first-level Hilbert curves (up to rotations) of half the size in each dimension. The black curve illustrates the third recursion level. All vertices are assigned to the closest grid point and assigned the corresponding the scalar value along the Hilbert curve, as depicted in Figure 12. This scalar value is known as a point's value in ***Hilbert space***. Since some vertices may be assigned to the same grid point, ties are broken in the ordering randomly. Thus, the vertices are processed in the order dictated by the Hilbert curve iterating through the 3D grid.

Fig. 9: The Russian street dog Laika is one of the first and most famous animals to travel through space.

Intuitively, one can see why the Hilbert curve might be a good ordering for the vertices by considering that mesh graphs are locally connected, meaning that the neighborhood of a vertex is typically nearby in 3D space. One well-known property of the Hilbert curve is that points that are close together in Hilbert space are also close in 3D space [5]. However, it is also true that randomly chosen points that are close together in 3D space are quite likely to be close in Hilbert space, as well [15, 18]. This leads to excellent cache behavior, since the neighbors of each vertex are close in 3D space for SIMIT-generated mesh graphs, and will thus tend to also be close in memory. We call the priority-dag scheduling algorithm with the Hilbert curve priority function LAIKA.[2]

In addition, the Hilbert curve has another convenient property that we can exploit toward the goal of partitioning a mesh graph in distributed memory. That is, a subinterval in Hilbert space corresponds to a compact subspace in ND space which has a low surface area to volume ratio [16, 17, 19]. Since mesh graphs are locally connected, we would then expect that the relative number of edges crossing from one such subspace to another would be low [15]. Thus, to distributed the computation among p different multi-core nodes in a distributed system, we merely split the vertices, presorted on the Hilbert priority function, evenly in p chunks while incurring relatively few inter-node messages. The use of space-filling curves for locality-preserving load-balancing is a well-known technique. Algorithms for the N-body problem [17, 19], database layout and scheduling [15], resource scheduling [12], and dynamic load balancing [6] all use variations on the general theme of mapping ND space onto a 1D curve that is subsequently partitioned among P processors.

1.4 Paper organization

We will explore the theoretical and experimental cache behavior of Hilbert-ordered data-graph computations of locally-connected mesh graphs in Section 2. In Section 3 we will describe a new scheduling algorithm LAIKA which exploits this cache advantage and test its performance in Section 3.1. We will explore the theoretical and experimental properties of an extension to LAIKA that enables distributed memory operation in Section 4.

2 Reordering Vertices for Cache Locality

[2] We take naming inspiration from the graph processing libraries GraphLab [13], which is named after a Labrador Retriever, and GraphChi [11], which is named after a Chihuahua. Laika, pictured in Figure 9, was a Russian street dog that was used in early space exploration [2] and we chose this name because Laika is a dog who travels through space, much like the Hilbert curve.

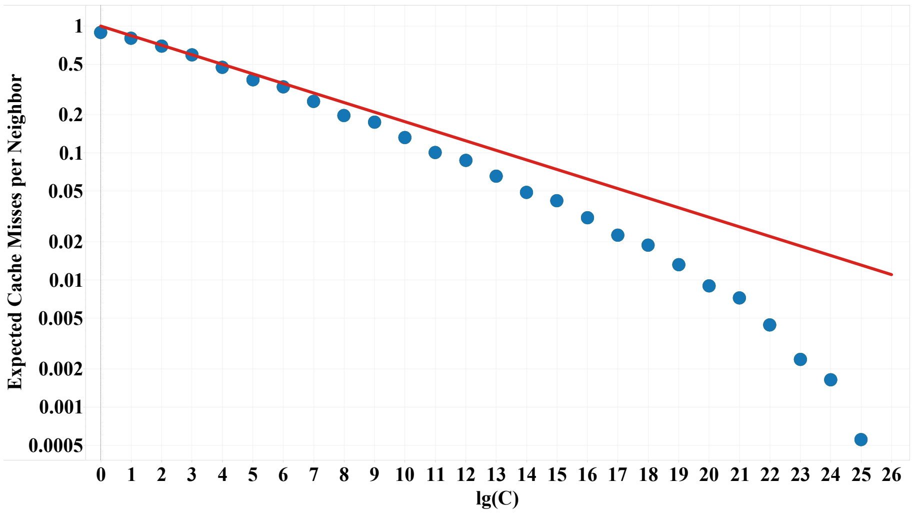

Fig. 10: Theoretical and empirically observed miss rate curves. The red line is generated by the equation $C^{-1/4}$, a consequence of the analysis in Tirthapura et al. [18] of generic recursively-defined space-filling curves. Hilbert curves are known to have better locality in practice. For example, the blue dots are empirically measured from a test graph with 50M vertices described in Section 3.1.

In this section, we will describe the rationale behind and empirical evidence supporting the use of the Hilbert space-filling curve as a way of mapping an N-dimensional space onto the real line. Specifically, we use this mapping to reorder the vertices of a 3D locally-connected graph to gain cache locality. Figure 12 illustrates a set of points on the unit square overlaid with a second-order 2D Hilbert curve. The ***Hilbert priority function*** for a vertex is equal to the value along the closest Hilbert curve grid point, breaking ties at random. The Hilbert priority function takes a parameter k which indicates the order of the Hilbert curve recursion, where one thinks of the curve dividing up a cubic space into 2^k x 2^k x 2^k blocks. This priority function is used to (comparison) sort the vertices in in-

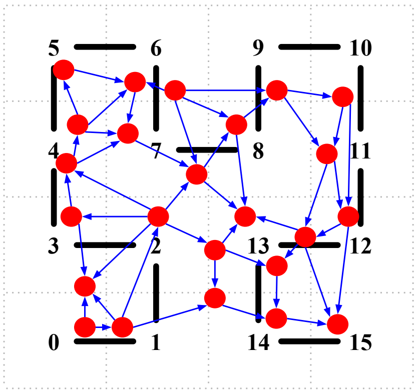

Fig. 12: Example of how a locally-connected graph in 2 dimensions is mapped to a dag via a second-order Hilbert priority function. Each vertex is mapped to its closest grid point in the discretized Hilbert curve. Among vertices mapping to the same Hilbert grid point, ties are broken randomly.

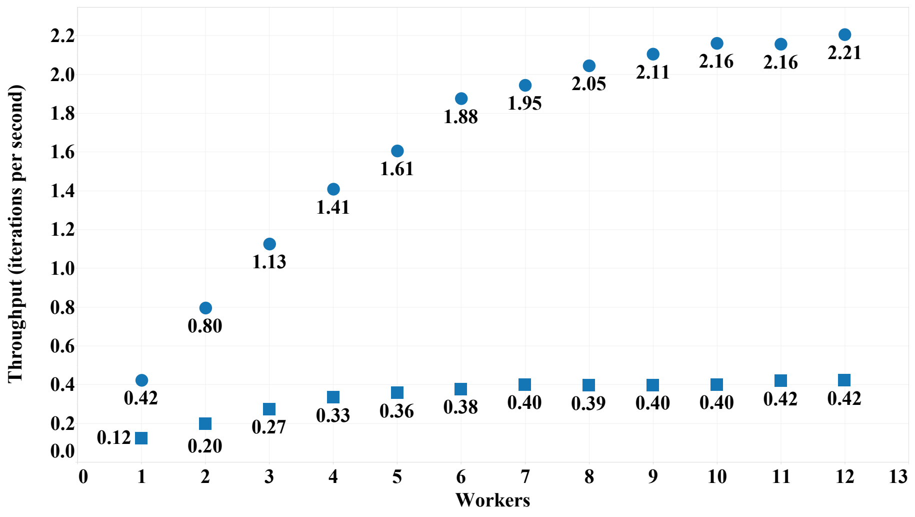

Fig. 11: Throughput vs. number of workers of the BSP scheduling algorithm, described in Figure 13, for the test graph with 50M vertices described in Section 3.1. The squares correspond to the test graph with randomly ordered vertices and the circles correspond to the same graph reordered according to the Hilbert priority function.

put graphs that are known to be locally-connected.[3]

We can see in Figure 10 that reordering the vertices according to the Hilbert priority function depicted in Figure 12 yields excellent cache behavior. For instance, with $\lg(C) = 11$ (i.e. 2048 vertices - roughly the size of the L2 cache in our Intel Xeon test system) only about 10% of the neighbors miss the L2 cache.[4]

The red line in Figure 10 is the cache miss rate predicted by an analysis in Tirthapura et al. [18]. They analyze a generic recursively-defined space-filling curve, which is pessimistic relative to the Hilbert curve as we see since our measured miss rates (i.e. the blue circles) appear asymptotically lower than the predicted values. Tirthapura et al. leave as an open problem the challenge of analyzing specific space-filling curves to achieve tighter bounds. They analyze the problem of dividing n vertices uniformly, generated on the unit cube and ordered by a space-filling curve, evenly among n^α ($\alpha \in [0, 1]$)

```
BSP($G$)
13  let $g = (V, E)$
14  **parallel for** $v \in V$
15      UPDATE($v$)
```

Fig. 13: The Bulk-synchronous Parallel (BSP) scheduling algorithm is the best-case scheduling algorithm in that all vertices are eligible at the beginning and thus BSP incurs the minimum possible scheduling overhead.

[3] For problems generated by Simit, there are generally sufficiently many time steps that any preprocessing, even $O(n \log n)$-time sorting, are completely amortized away.

[4] We use the notation $\lg(C)$ to mean $\log_2(C)$.

processors and finding all neighbors for each vertex within a distance r, which is set such that the average degree of the vertices is constant. They find that the total number of communications with other processors is $O(n^{(3+\alpha)/4})$. Since the average degree is constant there is $O(n)$ total work and thus the miss rate (i.e. the fraction of neighbors in another processor's vertex set) is $O(n^{-(1-\alpha)/4})$. Each processor is responsible for $n^{1-\alpha}$ vertices. An immediate consequence of the result of Tirthapura et al. is that with a cache of size $C = n^{1-\alpha}$, the expected miss rate is $O(C^{-1/4})$.

We use the BSP scheduling algorithm, detailed in Figure 13, to demonstrate the practical effect of reordering the vertices according to the Hilbert priority function. The BSP algorithm is quite simple: merely call UPDATE on all vertices in parallel. The Cilk work-stealing scheduler will tend to give each worker a contiguous run of vertices to execute and thus exploit the cache advantage inherent in the vertex reordering. In Figure 11 we see that the throughput achieved by the input graph reordered by the Hilbert priority function (i.e. circles) is much better in absolute terms than the throughput achieved by the original randomly ordered graph (i.e. squares).

3 Laika

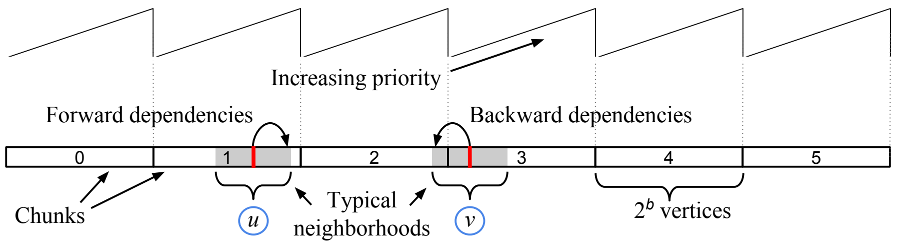

Fig. 14: Diagram describing the scheduling algorithm LAIKA detailed in Figure 15. The diagram consists of 2^b-vertex 'chunks', where b is a configuration parameter, each of which is processed serially. Due to the locality resulting from the Hilbert priority function, the neighbors of vertices labeled u and v, respectively, predominantly lie in the shaded regions surrounding them. Chunks are processed serially, so it is not necessary to keep track of predecessors and successors that lie within a single chunk. As a result, vertex u merely executes its update function and incurs no overhead for updating the counters of its neighbors, as with JONESPLASSMANN. The vertex v illustrates a common phenomenon, where vertices near the beginning of a chunk (i.e. chunk 3) have successors toward the end of the previous chunk (i.e. chunk 2), a *backward dependency*.

In this section we describe a new scheduling algorithm called LAIKA which takes advantage of the reordering of the vertices by the Hilbert priority function. It is a priority-dag scheduling algorithm in that each vertex has a counter which is decremented for each predecessor that is processed, however many of the (atomic) decrements are removed by design in LAIKA to reduce overhead. A diagram emphasizing aspects of LAIKA can be found in Figure 14 and pseudocode can be found in Figure 15. The algorithm works by breaking up the vertices into contiguous chunks of size 2^b, where b is a configuration parameter, and making progress on each chunk in rounds. In each round, the vertices within a chunk are processed serially and each chunk is processed in parallel. If at some point during the course of processing a chunk a vertex v is encountered which is not yet ready to be processed (i.e. $v.counter > 0$), then the current position in the chunk is recorded and progress is suspended until the next round. Due to the cache-locality property achieved by reordering, much progress is able to be made on each chunk, necessitating relatively few rounds.

```
LAIKA$(G, b)$
let $G = (V, E, \rho)$
**parallel for** $v \in V$
    $S = \{w \in v.adj : w.chunk \neq v.chunk\}$
    $v.pred = \{w \in S : \rho(w) < \rho(v)\}$
    $v.succ = \{w \in S : \rho(w) > \rho(v)\}$
    $v.counter = |v.pred|$
**parallel for** $i \in [0, |V|/2^b)$
    $chunkPointer[i] = i2^b$
$done$ = FALSE
**while** $done$ == FALSE
    $done$ = TRUE
    **parallel for** $i \in [0, |V|/2^b)$
        $p = chunkPointer[i]$
        **while** $p < (i+1)2^b$
            **if** $v_p.counter$ == 0
                UPDATE$(v_p)$
                **parallel for** $w \in v_p.succ$
                    $w.counter = w.counter - 1$
                $p = p + 1$
            **else**
                $done$ = FALSE
                BREAK
        $chunkPointer[i] = p$
```

Fig. 15: The Laika scheduling algorithm. The main **while** loop spins making progress on all chunks in parallel until all chunks have been completely processed. For each iteration, each chunk is processed serially until it is either done or encounters a vertex v which is not yet eligible to be processed (i.e. $v.counter > 0$). The priority function $\rho(v)$ returns the pair $\langle v.priority, v.chunk \rangle$ which is comparable lexicographically. The user-supplied function UPDATE is called for each vertex once all of its predecessors have been processed. The vertex state for the ith vertex v_i, $v_i.chunk = \lfloor i/2^b \rfloor$ and $v_i.priority = i \pmod{2^b}$ is defined at graph creation.

3.1 Multi-core Experimental Results

We performed several experiments to characterize the scalability of the BSP, LAIKA, and JONESPLASSMANN scheduling algorithms, the results of which are given in this section. We generated a random graph with 50M vertices uniformly distributed on the unit cube with an average degree of 16.4. Edges are formed by joining any two vertices within a distance $r = \sqrt[3]{3\Delta/(4\pi n)}$, where Δ is the average degree and n is the number of vertices. [5] This graph is used throughout this section as a proxy for the type of locally-connected graphs that we are likely to see from Simit. In future work, we will attempt

[5] The value of r follows from the fact that the expected number of neighbors Δ with n vertices in the unit cube is approximately $4\pi r^3 n/3$, the volume of a sphere of radius r times the number of vertices.

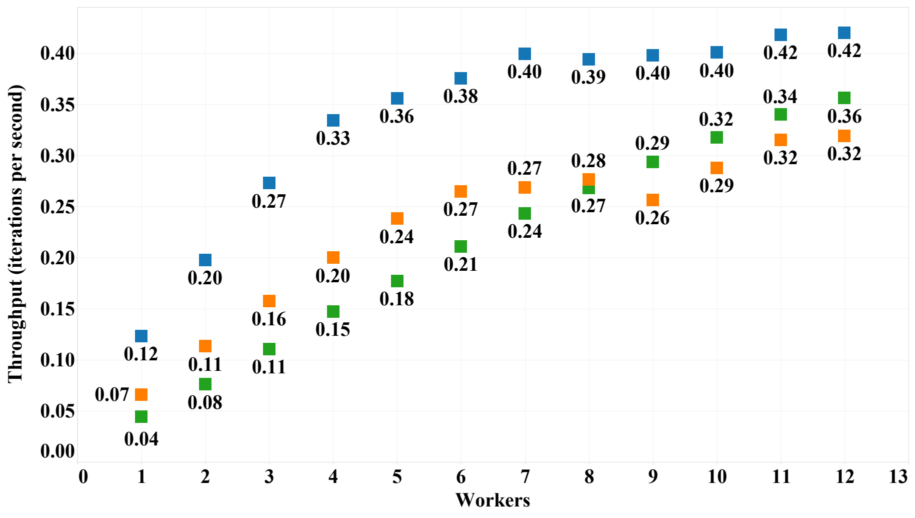

Fig. 16: Throughput vs. number of workers on the test graph with 50M vertices described in Section 3.1 for three scheduling algorithms: BSP (i.e. blue), LAIKA (i.e. orange) with $b = 16$ (i.e. 65,536 vertices per chunk), and JONESPLASSMANN (i.e. green).

to make synthetic generators of locally-connected 3D graphs which have less regular shape and density.

Figure 16 shows throughput as a function of the number of workers for the randomly ordered graph for the three scheduling algorithms: BSP (i.e. blue), LAIKA (i.e. orange), and JONESPLASSMANN (i.e. green). We can see that absent reordering with good caching behavior LAIKA is approximately equal to JONESPLASSMANN and scales worse. This is not surprising, as LAIKA is designed under the assumption that most of the edges in the graph occur between pairs of vertices within the same 2^b-vertex chunk.

Figure 17 demonstrates the caching advantage realized by reordering the vertices according to the Hilbert priority function with $k = 8$ (i.e. the unit cube is broken up into 2^k x 2^k x 2^k blocks). We see that LAIKA scales well and is within 10% of the relative ideal performance of BSP. Note that in order for BSP to be deterministic, the calculation must use double buffering of vertex state which would slow it down due to the extra memory usage, whereas the BSP timings in Figure 17 do not use double buffering and are thus not deterministic, but nonetheless serve as an approximate upper bound on performance of deterministic scheduling algorithms. Notice that JONESPLASSMANN is missing data for greater than 3 workers. In those cases, the experiments failed to complete due to an overflow of stack space. The recursive nature of JONESPLASSMANN can lead to problems of this sort when there are long dependency chains in the dag, which there are with reordering using the Hilbert priority function.

3.2 Effects of k^{th}-order Hilbert Reordering

In Figures 18 and 19 we see throughput as a function of k for serial and parallel executions, respectively. As k increases the granularity of the Hilbert curve shrinks, which

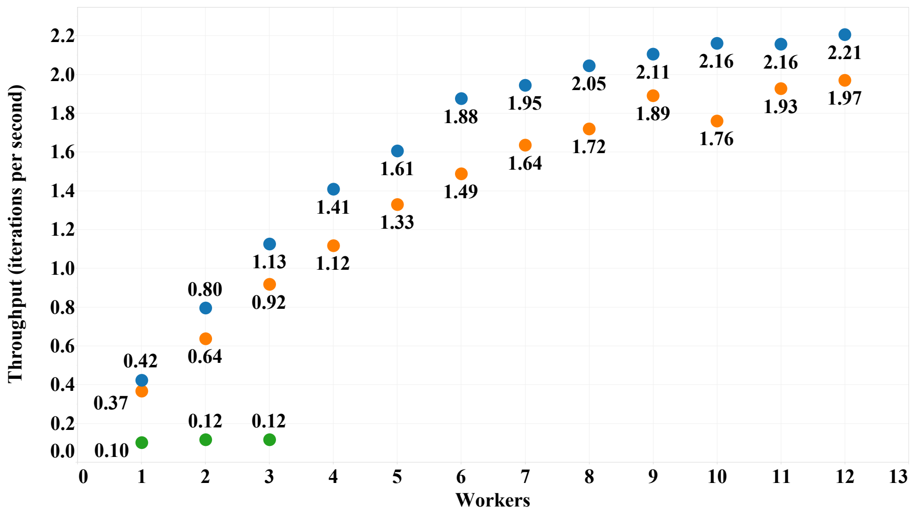

Fig. 17: Throughput vs. number of workers on the test graph with 50M vertices described in Section 3.1 reordered according to the Hilbert priority function for three scheduling algorithms: BSP (i.e. blue squares), LAIKA (i.e. orange squares), and JONESPLASSMANN (i.e. green squares).

increases the caching advantage at the expense of creating longer dependency chains. Each vertex has approximately 184 bytes of data associated with it (56 bytes in the vertex array itself and 128 bytes in the edge array). In Figure 18 we see a big jump in throughput between $k = 4$ and $k = 5$. At $k = 4$, there are 4096 total blocks traced out by the Hilbert curve, each of which has an average of 2.2MB of state (i.e. 184 bytes times 50M vertices divided by 4096 blocks). However, throughput jumps at the point $k = 5$ where the amount of state per block drops by a multiple of 8 or roughly the size of the L2 cache. We also notice that due to its depth-first nature of the recursive formulation, JONESPLASSMANN fails to take advantage of the increased cache locality offered by increasing the value of k. In fact, in Figure 19 we see that performance actually decreases with increasing k for JONESPLASSMANN, likely due to the fact that longer dependency chains reduce the parallelism in the computation.

4 Laika on Distributed Memory

In this section, we discuss a technique for executing LAIKA on a distributed memory system. First, we observe that reordering the vertices according to the Hilbert priority function, as discussed in Section 2, yields a data layout that is convenient for distributed memory, following from the same logic that gives it good cache locality. That is, the vertex set associated with each node is essentially a large cache such that as the memory footprint for each node grows the fraction of edges crossing between nodes shrinks. For instance, by the predicted cache miss rate from Tirthapura et al. [18], a node footprint of 16GB would result in approximately 1/100th of the edges crossing between nodes for any number of nodes. Intuitively, we can see why this is so in Figure 20, where

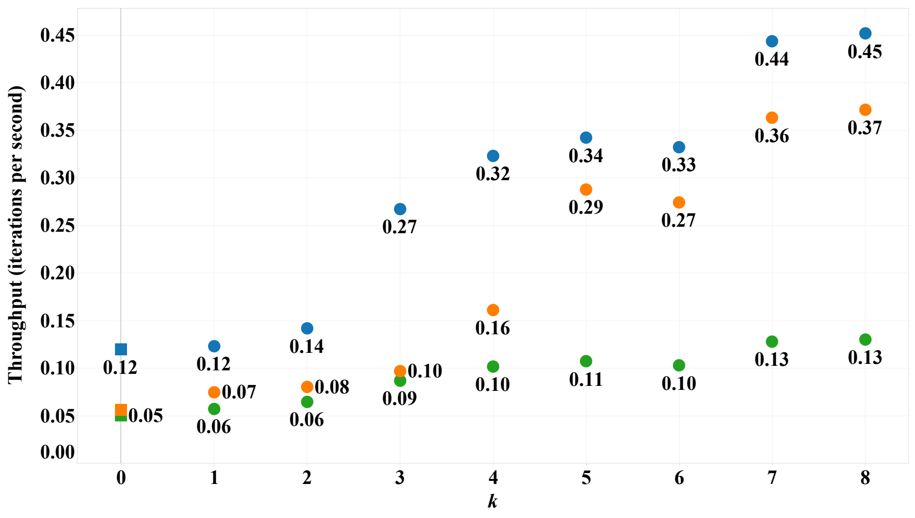

Fig. 18: Throughput vs. k on the test graph with 50M vertices described in Section 3.1 for three single-threaded scheduling algorithms: BSP (i.e. blue), LAIKA (i.e. orange), and JONESPLASSMANN (i.e. green). The value k is an input parameter to the Hilbert priority function, which discretizes the unit cube into a 2^k x 2^k x 2^k lattice where vertices lying within the same lattice block are ordered randomly. The special case of $k = 0$ (i.e. squares) indicates that the input graph consists of randomly ordered vertices.

we see an example of the general property that any subinterval of the Hilbert curve corresponds to a compact hypervolume in the corresponding ND space. This is clear from the construction of the Hilbert curve, as described in Figure 21, where each block in the Hilbert curve shares a face with an adjoining block. Since vertices are locally connected, we expect that the number of edge crossings is proportional to the (hyper) surface area of the volume, whereas the number of vertices is proportional to the actual volume, which is one dimension larger. As such, as the subinterval of the Hilbert curve grows the ratio of edge crossings to vertices shrinks.

4.1 Experimental Results

A simple extension to LAIKA accommodates execution in a distributed memory system using the message-passing interface (MPI) [1]. We merely divide up the vertices into R (for *ranks* in the MPI vernacular) contiguous regions and assign each to a rank. As a preprocessing step, one can find all of the edges in the edge set E that cross between ranks and execute the associated communication using MPI, instead of reads and writes in shared memory. We allocate a dedicated network thread who sends updated vertex state for a local vertex v to all ranks who own a vertex w such that $(v, w) \in E$. In addition, the network thread also receives updated vertex state for w and updates associated counters for all v such that $(v, w) \in E$ and w is a predecessor of v (i.e. $\rho(w) < \rho(v)$). Communication between the network thread and the workers is accomplished through a pair of concurrent queues [14].

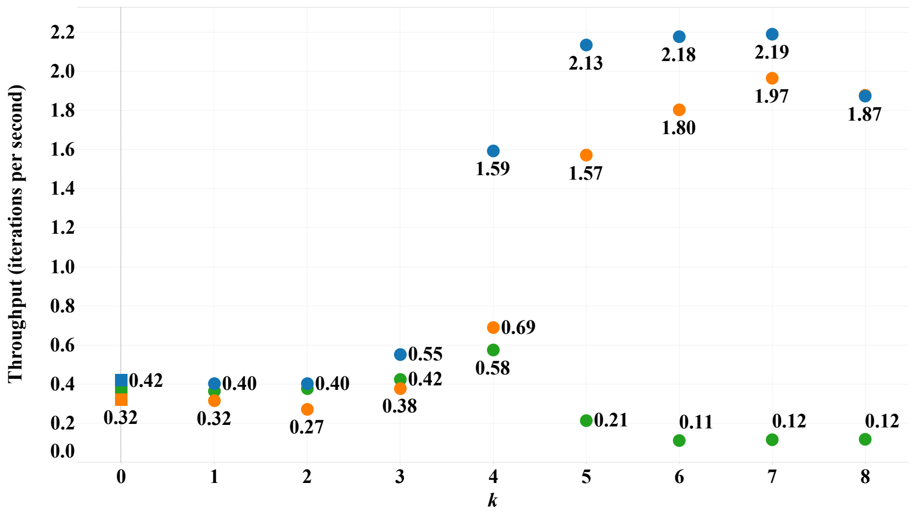

Fig. 19: Same as Figure 18 except that the scheduling algorithms are executed with 12 workers.

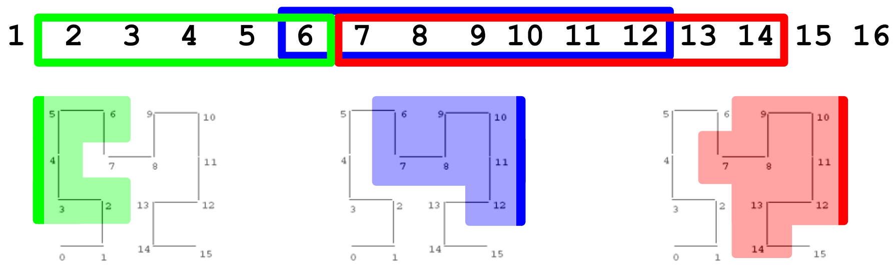

Fig. 20: Examples of how contiguous subintervals yield compact spaces in 2-dimensional space.

MPI Ranks	Time (s)
1	64.18
2	38.37
3	27.70
4	23.07
5	20.33
6	17.99
7	16.88
8	15.54

Fig. 22: Results of distributed memory experiment executing LAIKA extended using MPI on up to 8 individual 12-core Intel Xeons.

We tested this distributed memory version of LAIKA using a set of 8 Intel Xeons, each with 12 processor cores connected by gigabit ethernet. In particular, we performed a strong-scaling test of the 50M vertex graph described in Section 3.1. The strong-scaling results serve as a worst case for the typical scenario for Simit-generated graphs, which would use distributed memory primarily to execute on ever-larger graphs (i.e. weak-scaling). However, to give fair scaling results we adjusted the amount of work in the UPDATE function to reflect the typi-

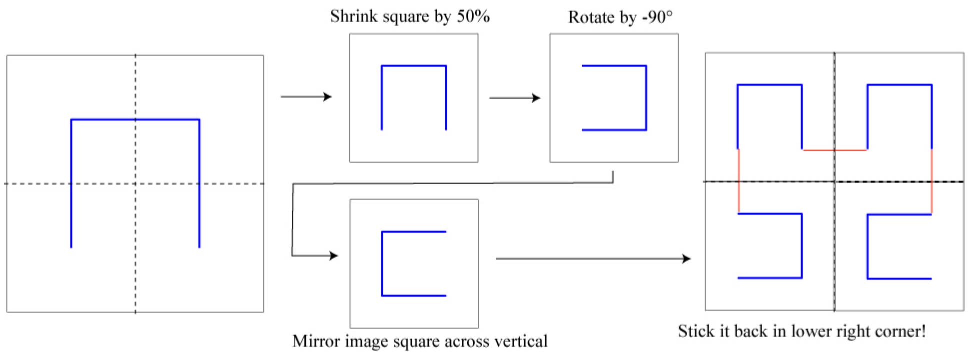
Shrink square by 50%
Rotate by -90°
Mirror image square across vertical
Stick it back in lower right corner!

Fig. 21: The construction of the Hilbert curve makes it clear that contiguous subintervals of the curve yields compact volumes in N-dimensional space: the curve always makes 90 degree turns in an N-dimensional construction, thus every pair of adjacent volumes in the Hilbert curve share a face.

cal amount of work in a Simit program, which generally performs a 3x3 matrix multiplication per neighbor in the graph. In Figure 22 we see the results of our experiment, where each rank is executed in paralles on 11 cores (i.e. the 12th core is used for the network thread) and we scale up to 8 multi-core ranks. We get approximately a 4x speedup on 8 ranks and a 2.78x speedup on 4 ranks. Relatively little performance engineering has been applied to this implementation, so we suspect that further gains are possible. In addition, we are applying this algorithm to a comparatively small graph of roughly 1GB. In practice, Simit will generate graphs that are approximately the size of main memory, which will also help our scalability.

References

1. MPI: a message-passing interface standard. Tech. rep., 1994.
2. Soviet fires new satellite, carrying dog; half-ton sphere is reported 900 miles up. New York Times, November 3, 1957.
3. Cutting, D., and Cafarella, M. Hadoop. `http://hadoop.apache.org/`, 2005.
4. Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM* (2008).
5. Gotsman, C., and Lindenbaum, M. On the metric properties of discrete space-filling curves. *IEEE Transactions on Image Processing* (1996).
6. Harlacher, D., Klimach, H., Roller, S., Siebert, C., and Wolf, F. Dynamic load balancing for unstructured meshes on space-filling curves. In *Proc. of the IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS) Workshops & PhD Forum, Shanghai, China* (2012).
7. Hasenplaugh, W., Kaler, T., Leiserson, C., and Schardl, T. B. Ordering heuristics for parallel graph coloring. In *SPAA* (2014).
8. Hilbert, D. Über die stetige abbildung einer linie auf ein flächenstück. In *Mathematische Annalen*. 1970.
9. Jones, M. T., and Plassmann, P. E. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing* (1993).

10. Kaler, T., Hasenplaugh, W., Schardl, T. B., and Leiserson, C. E. Executing dynamic data-graph computations deterministically using chromatic scheduling. In *SPAA* (2014).
11. Kyrola, A., Blelloch, G., and Guestrin, C. Graphchi: large-scale graph computation on just a pc. In *USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI, USENIX Association, pp. 31–46.
12. Leung, V. J., Phillips, C. A., Johnston, J. R., Arkin, E. M., Bender, M. A., Mitchell, J. S. B., Bunde, D. P., Lal, A., and Seiden, S. S. Processor allocation on cplant: Achieving general processor locality using one-dimensional allocation strategies. In *In Proc. 4th IEEE International Conference on Cluster Computing* (2002).
13. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment 5*, 8 (Apr. 2012), 716–727.
14. Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (1996).
15. Moon, B., Jagadish, H. V., Faloutsos, C., and Saltz, J. H. Analysis of the clustering properties of hilbert space-filling curve. Tech. rep., 1996.
16. Pilkington, J. R., and Baden, S. B. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Trans. Parallel Distrib. Syst.* (1996).
17. Singh, J. P., Holt, C., Hennessy, J. L., and Gupta, A. A parallel adaptive fast multipole method. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing* (1993).
18. Tirthapura, S., Seal, S., and Aluru, S. A formal analysis of space filling curves for parallel domain decomposition. In *ICPP* (2006).
19. Warren, M. S., and Salmon, J. K. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing* (1993).