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 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

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

software platform that performs these physical simulations in a fast and scalable manner on a distributed system of multi-core nodes.

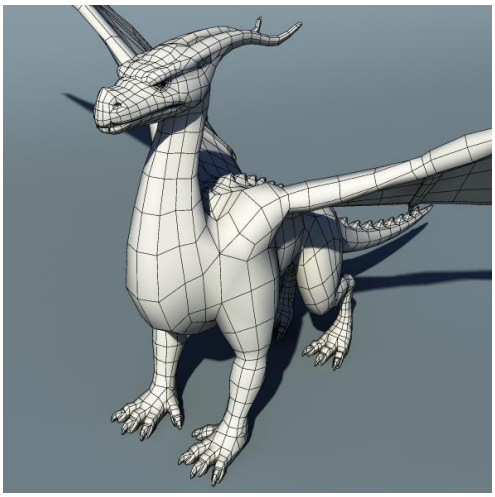

Fig. 1: A mesh graph where lines correspond to edges and intersections of lines correspond to vertices.

In recent years, there has been growing interest in developing frameworks for the storage and analysis of this data on large compute clusters, Hadoop [1, 2] 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, as we will explore throughout this paper.

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.

1.1 Data-graph Computations

In response to the shortcomings of applying Hadoop and similar systems to graph problems, Guestrin et al developed the GraphLab framework [6] for iterative graph algo-

rithms, 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

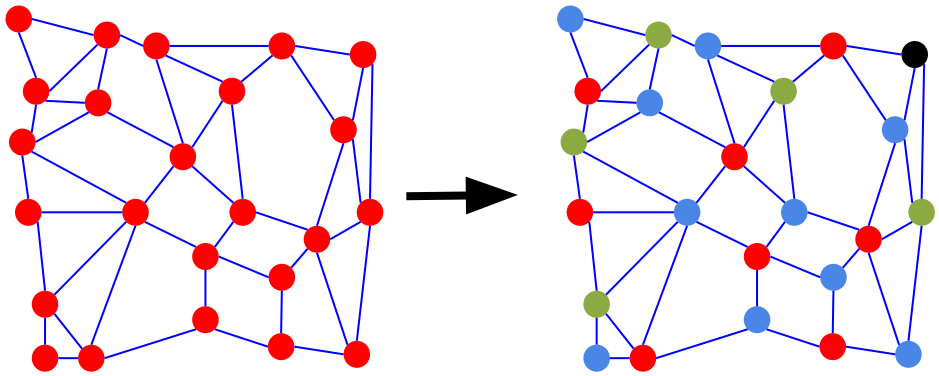

Fig. 2: Example of how a graph can be partitioned into independent sets of vertices denoted by color, each set of which is able to be executed simultaneously without causing data races. Iterating through the colors serially and executing the corresponding independent sets in parallel is a technique called *chromatic scheduling*.

Recently, Kaler et al [5] demonstrated that general data-graph computations could be made to be deterministic without giving up high-performance, in fact, while increasing performance. For instance, their system PRISM is 1.2-2.1 times faster than the non-deterministic 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 2, 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.

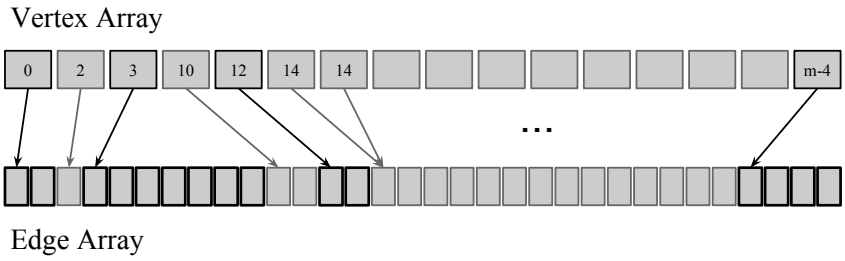

Fig. 3: Graphs are stored in memory on a single cache-coherent multi-core in a sparse-matrix format. The vertex array contains vertex data and an index into an edge array, which contains vertex IDs of the associated neighbors.

While chromatic scheduling does a good job of enabling high parallelism without any concurrency, it can be inefficient for cache usage. In Figure 3 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.

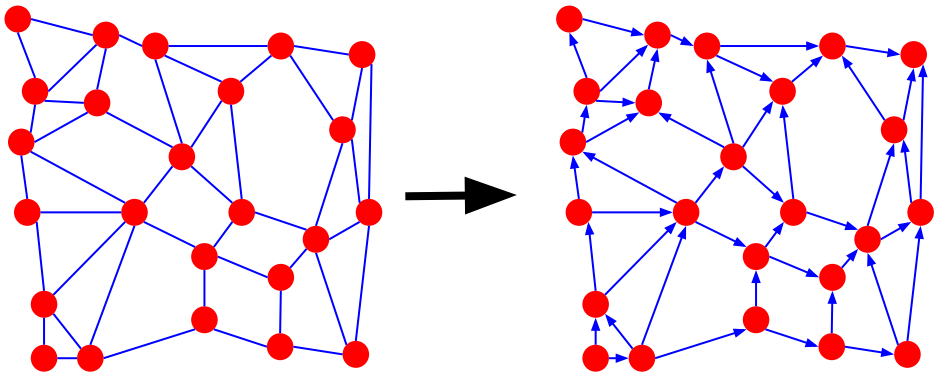

Fig. 4: An alternative to chromatic scheduling, which yields a deterministic, data race-free output, is dag scheduling. A priority funcrtion $\rho : V \to \mathbb{R}$ is used to create a partial order on the vertices, orienting an edge from low to high priority results in a dag. The vertices are processed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

An alternative approach to chromatic scheduling is ***dag scheduling*** [4], depicted in Figure 4 and used extensively by Hasenplaugh et al [3] 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 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 2.

1.2 SIMIT

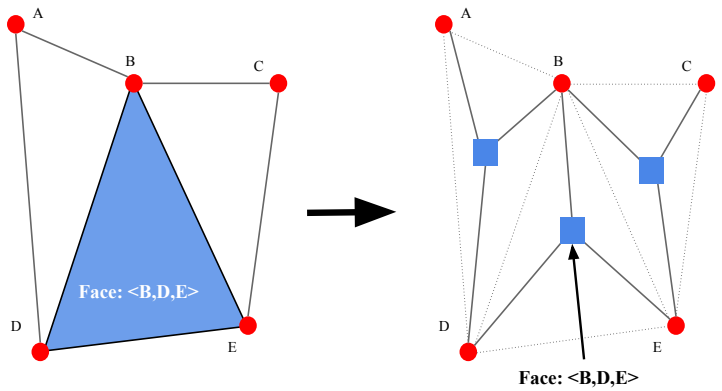

Fig. 5: Graphs generated by the language Simit have hyperedges, an example of which is in blue on the left. Hyperedges are represented by different *types* of vertices in the resulting data-graph computation. The square vertices in the figure represent hyperedges and have associated per-hyperedge data.

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 5 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 6 we see such an example, where a third type of vertex (i.e. the purple diamond in the figure) is 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.

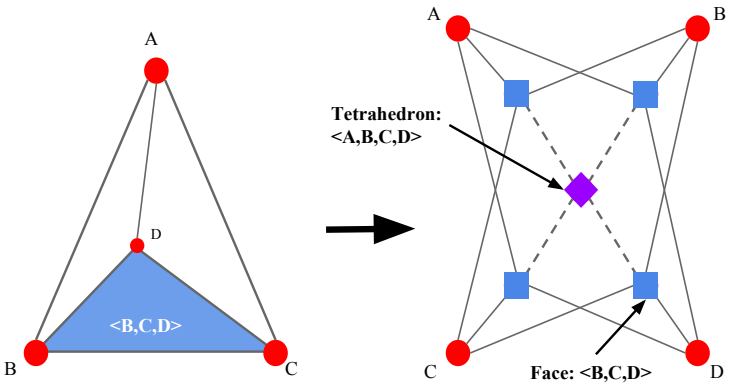

Fig. 6: Graphs generated by the language Simit have tetrahedrons, as depicted on the left above. A tetrahedron is composed of four hyperedges (or *faces*), an example of which is in blue on the left. Tetrahedra are represented by different *types* of vertices in the resulting data-graph computation. The diamong vertex on the right represents a tetrahedron and is connected to its four constituent hyperedges.

1.3 The Hilbert Space-filling Curve

In this paper we propose a new priority function for use with dag scheduling of data-graph computations generated by Simit. 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. A 2D example of the Hilbert space-filling curve is given in Figure 7. Then, all vertices are assigned to the closest grid point and assigned the corresponding the scalar value along the Hilbert curve, as depicted in Figure 8. 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.

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. 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. This leads to excellent cache

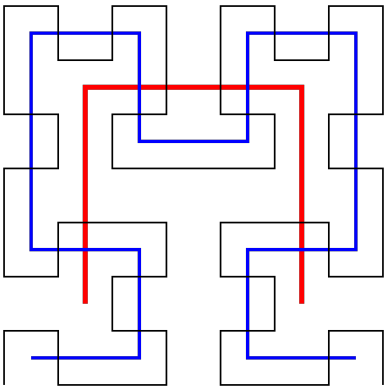

Fig. 7: Three recursion levels of a 2D Hilbert space-filling curve. The red curve is the first recursion level and illustrates the basic inverted 'U' shape. The blue curve shows how each quadrant is partitioned into four independent first-level Hilbert curves (up to rotations) of half the size in each dimension. The black curve illustrates the third recursion level.

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.

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. 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. Thus, to distributed the computation among p different multi-core nodes in a distributed system, we merely splic the Hilbert space evenly in p chunks, while incurring relatively few inter-node messages.

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 and Section 4, respectively. Then, we will explore the theoretical and experimental properties of the same class of mesh graphs, albeit much larger, in Section 5 and Section 7, respectively. Then, we will offer some concluding remarks in Section 8.

2 Space-filling Curves

Outline:

- Define hilbert priority function and general work flow of ordering algorithm
- Show distribution of vertex ID distance between every pair of neighbors w/ and w/o Hilbert ordering
- Give rationale for why it exhibits good cache behavior (perhaps show Theorem w/o proof)

In this section, we will describe the rationale behind using the Hilbert space-filling curve as a way of mapping an N-dimensional space onto the real line, specifically to use this mapping as a priority function for the priority-dag scheduler in PRISM. We will present event counters (e.g. cache misses, TLB misses etc.), measured performance and span for three serial experiments on the same set of graphs: chromatic scheduling, priority-dag scheduling with a random priority function and priority-dag scheduling with a Hilbert curve priority function.

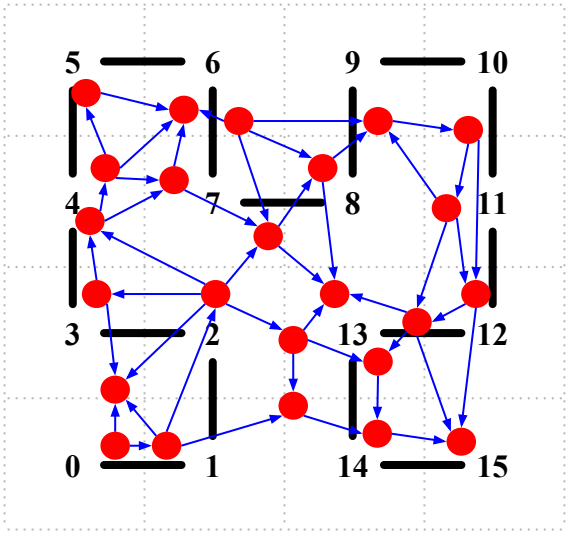

Fig. 8: Example of how a locally-connected graph in 2 dimensions is mapped to a dag via the Hilbert priority function. Each vertex is mapped to its closest grid point in the discretized Hilbert curve. Among vertices mapping to the same Hilbert grid point, ties are broken randomly.

3 Fast Execution on Individual Machines

Once we have a good vertex partitioning and a strategy for distributed execution, throughput depends largely on single-machine performance. We explore a number of schemes to this end.

3.1 Graph Representation in Memory

We represent graphs in memory on a single machine as follows. We have an array of vertices and an array of edges. Each vertex contains data and a pointer into the edge array indicating the start of its list of edges. Adjacent vertices have adjacent edge lists. Each edge is simply a pointer into the vertex array. This organization is shown in Figure 3.

3.2 Scheduling for Parallel Execution

There are two major strategies for scheduling vertices for parallel execution that preserve the appearance of a global ordering across updates and avoid data races. The first is coloring [**?**]. If we color a graph so that no two neighboring vertices have the same color, we can safely execute the updates for vertices of the same color in fully in parallel. The reason is that if any vertex's data is being written by some thread, it cannot be read in parallel by another thread because this would require a neighbor of this vertex to be updating concurrently, which is impossible. With the coloring strategy, we sort the vertex array (and the corresponding edge lists) by color and step sequentially through the colors, updating the vertices of each color in a parallel loop.

The other strategy is priority DAG scheduling [**?**]. This involves assigning each vertex a distinct priority, so that we can form a DAG from our graph by adding a direction to each edge such that the source is the endpoint vertex with higher priority. We assign each vertex a counter equal to the number of predcessors it has. We can start by executing all vertices with no predecessors in parallel. Once a vertex is complete, we atomically decrement the counter of each of its successors. If a vertex's counter becomes zero, we can spawn the update of this vertex as another parallel strand of execution. We thus attain fairly high parallelism at the cost of using atomics, which can involve expensive memory barriers on modern hardware. Once again, there can be no data races because two neighboring vertices cannot execute concurrently since one must be the predecessor of the other.

3.3 Achieving Cache Locality

If we update the vertices in the vertex array sequentially, as we do with coloring-based scheduling, we get good cache locality (cached lines are processed completely after being fetched) on our accesses to the vertices being updated and to their corresponding entries in the edge array, which is also processed sequentially. Cache locality here includes data cache and TLB locality, since pages in the vertex and edge arrays corresponding to vertices being updated are processed completely after their first access. TLB misses have been shown to be a significant factor in the runtimes of data-intensive computations and are an important consideration for us.

While we get good locality on edge and vertex array accesses for vertices being updated, we get poor locality for accesses to the vertex array to fetch these vertices' neighbors. These accesses are essentially random unless we have sorted the vertex array in some locality-improving manner. Ideally we could store vertices' neighbors close to them in the vertex array. This would confer two benefits. First, TLB misses would fall since neighbors of a vertex will in most cases be stored in the same page as the vertex. Secondly, if a vertex being updated pulled neighbors that were yet to be updated into cache, those neighbors would be updated before they left cache, reducing cache misses.

These considerations suggest that ordering vertices in the vertex array by breadth-first search (BFS) level would be helpful. A vertex at BFS level n can only have neighbors at BFS levels $n-1$ and $n+1$; if there was a neighbor at a smaller level, the vertex would have smaller level than n, and there can be no neighbors at levels greater than $n+1$ if the vertex is at n. Thus, if BFS levels are fairly small, ordering the vertices by

BFS level would result in nearby neighbor accesses as desired. BFS levels are generally bounded in size in mesh graphs; they grow at first, but once the largest cross-section of the mesh is reached, successive levels should have similar size. The problem with ordering the entire vertex array by BFS level is that there are no longer defined sequential regions over which parallel update loops can be run – since any two adjacent vertices could be neighbors – so parallelism is lost. We implement and evaluate a hybrid coloring-BFS approach that restores some parallelism while preserving our locality wins: within each BFS level, we sort by color, so that within each BFS level we can update the vertices of each color in parallel. The BFS levels are executed sequentially. This scheme brings up an important point about the locality-parallelism design space – after a point, increasing parallelism is not necessarily important. If there is enough parallelism to saturate the cores of the available machines, locality is likely the parameter worth optimizing.

In the case of priority DAG scheduling, cache behavior is very different. We lose the locality of access to the vertex and edge arrays for vertices being updated that we have in the sequential processing case. But we are not without victories: when we update the last predecessor of some node we immediately afterwards update that node, at which point is hot in cache. So accesses to neighbors of vertices being updated do not have worst-case cache behavior as they do in the sequential processing case.

For ideal cache behavior, the story is similar to the sequential processing case. We would like for neighborhoods of nearby vertices to be stored fairly contiguously in the vertex array so that accesses to neighbors of vertices being updated tend not to cause TLB misses. We would also like these neighborhoods to be updated completely in some small time window so that fetched neighbors are updated before they leave cache. Achieving the first objective is possible by using a BFS-based ordering or by preserving the Z-number ordering described above for partitioning vertices across machines. We take the second approach because BFS levels may be larger than memory pages and therefore TLB misses are more likely under the BFS-based ordering. Achieving the second objective is much harder with DAG scheduling because the order in which vertices are updated is unclear. However, if we assign each vertex a priority equal to its Z-number, we suspect that if we spawn off the processing of vertices with no predecessors in order of Z-number and if we can assume that earlier spawned routines tend to execute to completion before later spawned routines begin executing (as is the case in the Cilk model of multithreading), then contiguous regions of the physical mesh should be processed nearly to completion in small periods of time.

3.4 Considering Parallelism

3.5 Prefetching

4 Multi-core Experimental Results

Outline:

- Define experimental methodology
- Give results

5 Graph Partitioning for Distributed Memory

Outline:

- Rationale for why Hilbert ordering should be good for distributed memory
- Show fraction of edges that cross partitions w/ and w/o Hilbert ordering

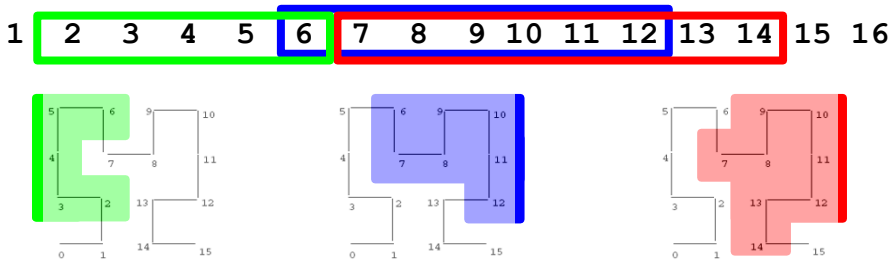

Fig. 9: Examples of how contiguous subintervals yield compact spaces in 2-dimensional space.

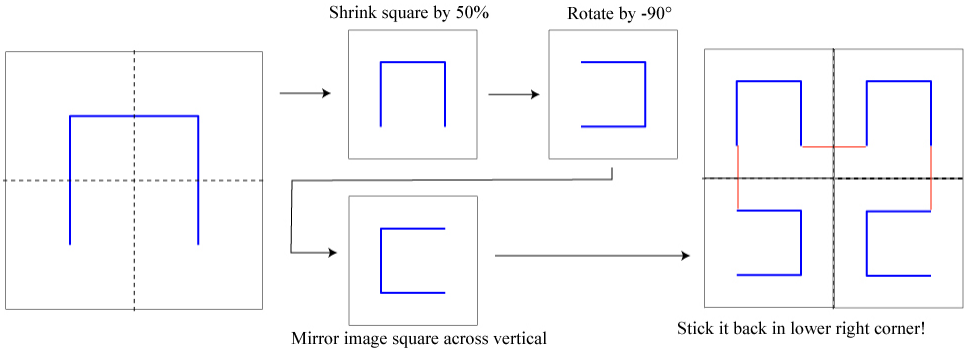

Fig. 10: The construction of the Hilbert curve makes it clear that contiguous subintervals of the curve yields compact volumes in N-dimensional space: the curve always makes 90 degree turns in an N-dimensional construction, thus every pair of adjacent volumes in the Hilbert curve share a face.

In this section, we will describe how the Hilbert curve is also a convenient mechanism for paritioning locally connected graphs that are embeddable in a low-dimensional space. That is, it generates a partition with small edge cuts. We will discuss how the priority-dag scheduling approach enables us to decompose the problem into two phases. The first phase is to extend PRISM to support a reshuffling operator, given a priority value from each vertex, which re-organizes the graph data structure in linear order according to the priority function. The second phase is to partition the n vertices by merely assigning n/p-sized compact subintervals of vertices to each of the p multi-core nodes. Finally, we will describe the software architecture that integrates MPI commands

communicating over edges spanning partitions with the priority-dag scheduled computations on each multi-core node. We will test the performance of our implementation by measuring strong-scaling performance on a small set of test graphs.

6 Dealing with Distributed Execution

Since an update function can only be run on a vertex if its neighbors are at most one iteration behind, we need to incur network traffic on every iteration to communicate vertex values for edges that cross machines. This can be done in a few ways. First, when an update function is being run on a vertex, it can fetch the values for neighbors on different machines and then run the necessary computation. This synchronous approach seems less than ideal, since the network traffic falls on the critical path of the iteration – it is very likely that vertices that are capable of being updated without any network requests are waiting idle as the network requests complete.

Thus, an asynchronous approach is generally preferable. Our approach is as follows. For each edge that crosses machines, we declare one endpoint vertex as the predecessor and the other as the successor. Once the predecessor is updated, it sends its value to the machine to which the successor is assigned. Once the successor has received values from all of its predecessors, it can be scheduled for execution. If our partioning of vertices across machines is good and there are many vertices on each machine that have no dependencies on external vertices and therefore can be scheduled at any time, then there will be minimal waiting for network messages. Our scheme ensures that we satisfy the constraint that vertex updates must appear to be processed in some global order – in other words, both of the endpoints of an edge cannot be updated in some iteration based on the other's data from the previous iteration.

We partition vertices across machines using a technique based on space-filling curves. A 3D space-filling curve maps the real numbers to points in 3D space so that for an arbitrary point p as we trace out more and more of the curve the distance from p to the nearest point on the curve becomes smaller and smaller. We use a Z-order curve, which has the property that if two points on the curve are relatively close together, then the real numbers that generated those points tend to be relatively close. A Z-order curve in 3D space is shown in Figure 2.

Each vertex in a mesh graph can be assigned a particular coordinate in 3D space. Thus, we can draw a bounding box in 3D space around a mesh graph. We trace out a Z-order curve over a finite portion of its domain mapping to points n the bounding box. We then assign each vertex in the mesh to its nearest point in the traced curve and mark the vertex with the real number that generated this point, which we will call the Z-number. We then sort the vertices by Z-number. Nearby vertices in the sorted order should be nearby in the physical graph. We can then split this sorted list into contiguous chunks, one for each machine in our cluster. Since each chunk should correspond to some contiguous region in 3D space, the number of edges crossing machines should be fairly small, as explained in the previous section.

7 Distributed Memory Experimental Results

Outline:

- Define experimental methodology
- Give results

8 Conclusion

References

1. CUTTING, D., AND CAFARELLA, M. Hadoop. `http://hadoop.apache.org/`, 2005.
2. DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM* (2008).
3. HASENPLAUGH, W., KALER, T., LEISERSON, C., AND SCHARDL, T. B. Ordering heuristics for parallel graph coloring. In *SPAA* (2014).
4. JONES, M. T., AND PLASSMANN, P. E. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing* (1993).
5. KALER, T., HASENPLAUGH, W., SCHARDL, T. B., AND LEISERSON, C. E. Executing dynamic data-graph computations deterministically using chromatic scheduling. In *SPAA* (2014).
6. 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.