Fast Iterative Graph Computation with Block Updates

Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, Johannes Gehrke
Cornell University
Ithaca, NY
{wenleix, guoz, bindel, ademers, johannes}@cs.cornell.edu

ABSTRACT

Scaling iterative graph processing applications to large graphs is an important problem. Performance is critical, as data scientists need to execute graph programs many times with varying parameters. The need for a high-level, high-performance programming model has inspired much research on graph programming frameworks.

In this paper, we show that the important class of *computationally light* graph applications – applications that perform little computation per vertex – has severe scalability problems across multiple cores as these applications hit an early "memory wall" that limits their speedup. We propose a novel block-oriented computation model, in which computation is iterated locally over blocks of highly connected nodes, significantly improving the amount of computation per cache miss. Following this model, we describe the design and implementation of a block-aware graph processing runtime that keeps the familiar vertex-centric programming paradigm while reaping the benefits of block-oriented execution. Our experiments show that block-oriented execution significantly improves the performance of our framework for several graph applications.

1. INTRODUCTION

Graphs express complex data dependencies among entities, so large graphs are a key modeling component for many applications, such as structure from motion [9], community detection [32], physical simulations [33], and link analysis [7]. Graph processing usually exploits parallelism, since it is both compute- and memory-intensive. Recently several graph computation frameworks have been introduced with the express goal of helping domain experts develop graph applications quickly [27, 25, 26, 19, 12, 40]. These frameworks present to their users a "think-as-a-vertex" programming model in which each vertex updates its own data based on the data of neighboring vertices. The programming model is coupled with an iterative execution model, which applies the vertex update logic repeatedly until the computation converges. These frameworks have been used successfully in many graph processing applications [27, 25, 19].

Different graph applications perform different amounts of computation per vertex. *Computationally light* applications, such as

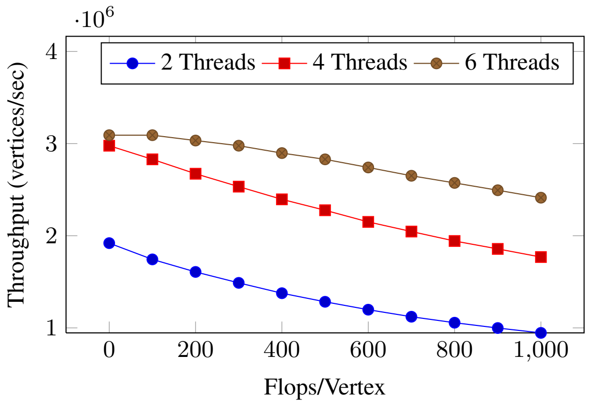

Figure 1: Memory Wall for Lightweight Computation

PageRank, perform tens of floating point operations (flops) per vertex, whereas *computationally heavy* applications such as Belief Propagation, may perform orders of magnitude more work per vertex. This significantly affects performance: A computationally heavy application can fully utilize several cores, while computationally light applications are limited by memory access rates.

We illustrate this phenomenon by running GRACE, our highly-optimized graph processing engine [40], on a Xeon machine with two sockets with four cores each. The processing threads are evenly distributed across sockets, and there are no bottlenecks due to locking or critical sections [40]. Figure 1 shows how computational load, measured in flops per vertex, impacts throughput, measured in vertex updates per second. For computationally heavy applications, adding threads significantly improves throughput. But for computationally light applications, there is negligible improvement in throughput beyond four threads. This is because GRACE has reached the memory bandwidth of the processor; the time spent retrieving vertex and edge data from memory exceeds the time spent computing on the data. Adding more cores only exacerbates this problem.

In this paper, we propose a novel block-oriented computation modelthat significantly improves the throughput of computationally light graph applications. Inspired by the block-oriented computation model for matrices and grids from the HPC community [4], we use standard methods to partition the graph into blocks of highly connected vertices [3, 20, 36, 12]. Then, instead of scheduling individual vertices, we schedule blocks of related vertices. This new model opens up two opportunities. First, we can update a block repeatedly to improve locality while accelerating convergence. Repeatedly updating one vertex does not improve convergence, as the

Proceedings of the VLDB Endowment, Vol. 6, No. 14
Copyright 2013 VLDB Endowment 2150-8097/13/14... $ 10.00.

neighboring vertices always provide the same data, and thus the computation always produces the same result. But in repeatedly updating a block of connected vertices, each step can make additional progress. Second, we can use a different scheduler inside a block than the one we use across blocks. For example, we may schedule nodes within blocks in a round-robin fashion, but schedule blocks based on which block contains the node with the worst residual error. Since updating a block is more expensive than updating one vertex, we can afford the overhead of a relatively more expensive scheduler to choose blocks. We show that different scheduling algorithms (of different cost) significantly affect the convergence of graph computations, and thus the overall performance.

We have added support for block-level computation to GRACE through a new block-aware runtime. This runtime has a novel block-level concurrency control protocol based on snapshot isolation; our approach effectively minimizes concurrency overhead. The runtime supports separate scheduling policies for blocks and for nodes within blocks, allowing users to trade off scheduling cost against speed of convergence.

In the next section, we introduce the vertex-centric programming abstraction for iterative graph processing and demonstrate the scalability problems associated with poor locality in the corresponding vertex-oriented computation model. In Section 3 we introduce our block-oriented computation model, which has better locality. In Section 4, we turn to our block-aware execution engine, and describe in detail the scheduling mechanisms that allow us to maintain fast convergence with low scheduling overhead. We present experimental results in Section 5 to demonstrate how our block-oriented computation model can improve update throughput and convergence rates for real-world graph processing applications. We survey related work in Section 6 and conclude in Section 7.

2. BLOCK VS. VERTEX-AT-A-TIME

Most parallel graph processing frameworks present to their users a vertex-centric programming abstraction: application logic is written as a local update function to be run at individual vertices. This function is "local" in the sense that it is executed on a single vertex, and updates the vertex data as a function of data at neighboring edges and vertices. In this model, vertices communicate only with their neighbors, either by sending messages or by remote data access. Thus, vertex updates can proceed in parallel, with low-level details of the parallelism handled transparently by the framework.

To be more concrete, suppose we are given a directed graph $G(V, E)$. Users can define arbitrary attributes on vertices and edges to represent application data. To simplify the discussion, we assume vertex values can be modified, but edge values are read-only. This assumption does not limit expressiveness, since we can store the writable data for edge (u, v) in vertex u.

We denote the data on vertex v by S_v, extending this notation to sets of vertices as well. Similarly, we denote the data on edge (u, v) by $S_{(u,v)}$. The update function for a vertex v depends only on data on its incoming edges $NE(v)$, and on the vertices $NV(v)$ that connect to v through $NE(v)$. The function $\mathsf{VertexUpdate}$ that maps the current state S_v^{old} of vertex v to its new state S_v^{new} has the following signature:

$$S_v^{\text{new}} = \mathsf{VertexUpdate}(S_v^{\text{old}}, S_{NV(v)}, S_{NE(v)}).$$

During execution, the runtime schedules individual vertex updates. To achieve scalability, existing frameworks either (1) follow the Bulk Synchronous Parallel (BSP) model [38], arranging updates into iterations separated by global synchronization barriers, with updates in one iteration depending only on data written in the previous iteration [27, 19, 42, 37, 40]; or (2) update vertices asynchronously, based on the most recent data from neighboring vertices, with static or dynamic scheduling of the updates to achieve fast convergence [28, 26, 22]. Algorithm 1 summarizes this vertex-oriented computation model, in which different scheduler implementations can lead to either synchronous or asynchronous execution policies. Whatever scheduler is used, the resulting execution policy is at the granularity of vertices: the processor accesses one vertex at a time, loading the data from the vertex and its neighbors into local cache and triggering the update function $\mathsf{VertexUpdate}$. Note for the BSP model the commit is not executed immediately but logged, and will be executed at the synchronization barrier.

Algorithm 1: Vertex-Oriented Computation Model

```
1 Initialize the vertex data ;
2 repeat
3     Get a vertex $v$ to be updated from the scheduler ;
4     Update the data of $v$ based on $S_{NV(v)}$ and $S_{NE(v)}$ ;
5     Commit the update to $S_v$ ;
6 until No vertex needs to be updated anymore;
```

The vertex-centric model is a useful programming abstraction, but the corresponding vertex-centric update mechanisms result in poor performance for computationally light graph algorithms, such as PageRank, shortest paths, connected components, and random walks. Such algorithms are communication-bound: computing a vertex update is cheaper than retrieving the required data from neighboring vertices. Thus, these algorithms scale poorly with increasing parallelism. Researchers have proposed ways to reduce the networking overhead for computationally light applications in distributed memory environments [12, 18]; but we have observed poor scaling even in shared-memory environments, where main memory bandwidth quickly becomes a bottleneck [40].

On the other hand, researchers in high-performance computing (HPC) have studied communication bottlenecks for many years in the context of sparse matrix routines and PDE solvers, which make up a special class of iterative graph processing algorithms [4]. Many of the optimization techniques from this literature apply to other graph algorithms as well.

Domain Decomposition for Elliptic PDEs. Large, sparse linear systems and non-linear systems of equations often come from discretized elliptic partial differential equations (PDEs). These linear systems are typically solved by iterative methods [4], the most common of which are Krylov subspace methods with a preconditioning solver to accelerate the rate of convergence. For these problems, domain decomposition is widely used to construct both preconditioners and stand-alone solvers [34]. Domain decomposition methods partition the original domain into disjoint or overlapping subdomains, possibly recursively. The subdomain sizes are typically chosen to minimize inter-processor communication and maintain good cache locality. Each subdomain is updated in a local computation. Because the subdomain solvers are often expensive, these methods tend to have good communication-to-computation ratios; consequently, domain decomposition methods are popular in parallel PDE solvers.

Block-oriented Scheduling for Eikonal Equations. The eikonal equation is a nonlinear hyperbolic PDE that describes continuous shortest paths through a medium with varying travel speeds. It is used in many applications, ranging from optics to etch simulation [33]. Various methods with different vertex-scheduling mech-

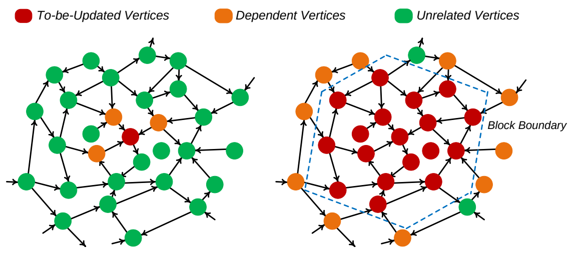

Figure 2: Vertex- vs. Block-Oriented Computation

anisms have been proposed for this problem. Fast-Marching methods, like Dijkstra's algorithm for discrete shortest path problems, dynamically schedule nodes so they are processed in increasing order of distance from the starting set. In contrast, Fast-Sweeping methods update in a fixed order, and so have lower scheduling overhead and more regular access patterns; but they require multiple iterations to converge. As with elliptic PDEs, domain decomposition has been used to parallelize Fast-Sweeping [43]; and recent work has introduced a domain decomposition approach that uses sweeping on subdomains and marching to schedule subdomain solves [8]. It has proven to be very effective, achieving fast convergence rates by using dynamic scheduling for subdomains while maintaining low scheduling overhead by using static scheduling at the per-vertex level.

Blocking is a standard idea that has been applied in many settings. Examples include tile-based Belief Propagation [24], Block Coordinate Descent [6], and cache-aware graph algorithms [30]. Unfortunately, a general data-centric framework for graph structured computations, which would save people from reinventing the wheel, is missing from the literature. In this paper, we take the first step toward this goal by proposing a general block-oriented computation model for graph computations. As we will show below, our block-oriented computation model still works with a vertex-centric programming abstraction to achieve easy programming.

3. BLOCK-ORIENTED COMPUTATION

The block-oriented computation model is a natural extension to the vertex-oriented computation model. Figure 2 illustrates the two models: small red, yellow, and green circles represent vertices to be updated, vertices on which the update depends, and vertices unrelated to the current update. In the vertex-oriented computation model, only one vertex and the data on which it depends are loaded from memory for each update, while in the block-oriented computation model, all the vertices belonging to the same block are loaded from memory and updated together. For a cluster of processors, one can first partition the graph and assign subgraphs to the processors, then further partition the assigned subgraphs into blocks. The subgraphs are chosen to minimize the number of edges between them, so that adjacent vertices are likely to be in the same block.

3.1 Block Formulation

We initially partition $G(V, E)$ into disjoint blocks $B_1(V_1, E_1)$, $B_2(V_2, E_2)$, $\cdots$, $B_k(V_k, E_k)$, where k should be much greater than the expected degree of parallelism. Block B_i contains all the edges that originate in V_i, i.e. $E_i = \{(u, v) \in E | u \in V_i\}$. Vertices V_i are the *local vertices* of B_i; and edges E_i are its *local edges*.

Algorithm 2: Block-Oriented Computation Model

```
1  Initialize the graph data ;
2  repeat
3      Get a block $B$ to be updated from scheduler ;
4      Update the data of $u$ based on its boundary data $S_{NV(B)}$
       and $S_{NE(B)}$ ;
5      Commit the update to $S_{V(B)}$ ;
6  until No block need to be updated anymore;
```

Algorithm 3: Cache-Aware Vertex-Oriented Computation Model

```
1  Initialize the vertex data ;
2  repeat
3      Get a set of vertices $V' \subset V$ from global scheduler, the
       vertices in $V'$ are closely connected ;
4      Add the scheduled vertices in $V'$ into the local queue $Q$ ;
5      while Local queue $Q$ is not empty do
6          $v = Q$.pop() ;
7          Update the data of $v$ based on $S_{NV(v)}$ and $S_{NE(v)}$ ;
8          Commit the update to $S_v$ ;
9      end
10 until No vertex needs to be updated anymore;
```

For a given block B, we will also use $V(B)$ to denote its set of local vertices and $E(B)$ to denote its set of local edges. When updating B_i, the local edges are read-only and the local vertices are read-write. We define the *incoming boundary vertices* of B_i as $NV(B_i) = \{u \in V - V_i | (u, v) \in E, v \in V_i\}$; and we define the *incoming boundary edges* as $NE(B_i) = \{(u, v) \in E | u \in V - V_i, v \in V_i\}$. This part of the state is read-only when updating B_i, and can be viewed as part of the input to the update procedure for B_i. Block B_j is an *incoming neighbor block* of B_i if B_j contains any incoming boundary vertices of B_i, i.e. $V_j \cap NV(B_i) \neq \emptyset$

Similarly, we define the *outgoing boundary vertices* of B_i as $OV(B_i) = \{v \in V - V_i | (u, v) \in E, u \in V_i\}$, and the *outgoing boundary edges* as $OE(B_i) = \{(u, v) \in E | v \in V - V_i, u \in V_i\}$. Block B_j is an *outgoing neighbor block* of B_i if $V_j \cap OV(B_i) \neq \emptyset$

The goal of the partitioning is to minimize the number of edges cut by the partition, while making the blocks roughly the same size to facilitate load balance. This is a well-studied problem for which many efficient methods are known [20, 3, 36, 35].

3.2 Per-Block Update

We organize computations around a *block update function*; see Algorithm 2. The block update function has the signature

$$S_B^{\text{new}} = \mathsf{BlockUpdate}(S_B^{\text{old}}, S_{NV(B)}, S_{NE(B)}),$$

where $S_{NV(B)}$ and $S_{NE(B)}$ denote the data on the boundary vertices and boundary edges of B. A straightforward way to implement this function would be to let the user directly specify the block-level logic: by analogy to the vertex programming abstraction, we could have a *block programming abstraction*, exposing the block structure, block data, and the dependent boundary data to the user. However, this block programming abstraction would not follow the "think-as-a-vertex" philosophy that has proven so successful in practice [27, 12]. The block-centric programming abstraction is more complicated than the vertex-centric programming abstraction because it introduces an artificial distinction between local and boundary vertices: a local vertex is modifiable and can access its

Table 1: Benefit of Cache Performance

Scheduler	Time	# Updates	# LLC Misses
Non Cache-Aware	9.52	34,152,807	197,500,000
Cache-Aware	5.15	34,152,807	37,500,000

neighbors' data, while a vertex vertex is read-only and has no access to its neighbors' data. Moreover, users are already familiar with the vertex-centric programming abstraction; many graph applications have already been written using it, and it would be inconvenient for users to learn a new abstraction and migrate their existing applications into it. Thus our goal is to let users write vertex-centric programs, then run those programs in the block execution model. To achieve this, our block update function is defined in terms of a (traditional) vertex update function and an optional inner-block scheduling policy $\mathsf{InnerScheduler}$:

$$\mathsf{BlockUpdate} = \mathsf{InnerScheduler}(\mathsf{VertexUpdate})$$

The inner-block scheduler iterates over some or all of the vertices inside a block and applies the user-specified $\mathsf{VertexUpdate}$ function to these vertices, possibly multiple times. For example, a simple inner-block scheduler could update each vertex in the block exactly once in a fixed order, while a more sophisticated scheduler could update vertices repeatedly until the block data converged.

A key benefit of this block computation model is improved locality of reference: by loading a block of closely connected vertices into the cache, we increase the cache hit rate and thus reduce the average data access time. We demonstrated this effect in the GRACE engine [40] using the experimental setup described in detail in Section 5.2. We ran personalized PageRank on the Google graph using both a vertex-centric scheduler (Algorithm 1) and a cache-aware scheduler (Algorithm 3). Algorithm 1 uses the default vertex scheduler in GRACE: it iterates over the vertices in a random order until convergence, with no regard for graph partitioning. The cache-aware scheduler in Algorithm 3 uses the graph partitioning to improve cache performance. Instead of fetching one vertex v and updating it, the cache-aware vertex scheduler fetches a set V' of closely-connected vertices, then updates each vertex in turn. Since vertices in V' share many neighbors, this method achieves better cache utilization. As shown in Table 1, the cache-aware vertex scheduler reduces the number of Last-Level-Cache (LLC) misses by about 80% compared to the non-cache-aware vertex scheduler, and reduces the total run time by nearly 50% with the same number of updates. We observed this effect in all the social graphs in our experiments.

Computationally light applications, which exhibit high data access to computation ratios, can run even faster by updating each vertex in a block multiple times before evicting it from the cache. This reduces the data access to computation ratio, improving end-to-end performance as long as the extra computations make at least some progress towards convergence. In fact, this idea has been used for years in large-scale linear system solvers in the high-performance computing literature [34]. We illustrate this idea again on the Google graph by showing the overall time for ten steps of the computation of personalized PageRank. Figure 3 shows how block updates improve the data access to computation ratio. On one thread, each extra sweep over the block increases the overall time by about 530 ms, while on eight threads, which use much more memory bandwidth than one thread, each extra sweep takes only about 20 ms. Thus, for computationally light applications updating data in the cache is cheap. As the amount of parallelism increases and each thread's share of the available memory bandwidth decreases, we should iterate over vertices that are already in cache multiple times as long as this accelerates convergence.

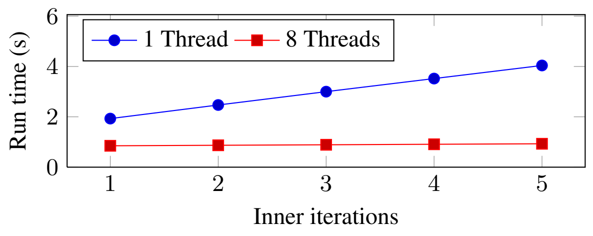

Figure 3: Running time vs. inner iterations for PageRank.

3.3 Two-Level Scheduling

In addition to its improved cache behavior resulting from data locality, our block-oriented computation model also enables flexible scheduling of computation both at the block level and within each block. As discussed in Section 2, dynamic scheduling of vertex updates can improve convergence. However, this improvement comes at a cost. A vertex-oriented dynamic scheduler usually requires some scheduling metadata on each vertex, such as a scheduling priority value or a flag indicating whether the vertex is in the scheduling queue. Updating this metadata and querying it to make scheduling decisions can be expensive relative to a simple static scheduler that stores no scheduling metadata. For example, a prioritized scheduler maintains priority values for all the vertices and selects the highest priority vertex to be processed next. A common implementation uses a heap-based priority queue, and requires $\Omega(\log |V|)$ time to update the vertex priority and to pop the highest priority vertex from the queue.

As a result, even if a dynamic scheduler performs fewer vertex updates than a static scheduler, the dynamic scheduler might yield worse overall performance due to the extra scheduling overhead. For example, the Fast-Marching method can be outperformed by the simpler Fast-Sweeping method on problems with constant characteristic directions, because the dynamic scheduling overhead of the Fast-Marching method outweighs its benefit of triggering fewer vertex update functions [8].

For the block-oriented computation model, however, scheduling decisions can be made at the block level instead of the vertex level. This greatly reduces the overhead of dynamic scheduling, since the scheduling metadata only needs to be maintained per-block. Making scheduling decisions at the block level can be less accurate than making them at the vertex level, and thus result in more vertex updates before convergence, but for computationally light applications this is unlikely to be a problem, since the number of vertex updates alone does not determine the overall performance.

In addition to the block-level scheduler, which chooses the order in which blocks are updated to achieve fast global convergence, we may benefit from an inner-block scheduler that chooses the order in which vertices are updated within the scheduled block. Although high overheads make vertex-level dynamic inner-block schedulers unattractive for computationally light applications, some applications still benefit from various static inner-block schedulers. For example, alternating sweeping ordering within the block can improve the convergence of the fast sweeping method and the Bellman-Ford algorithm [21, 8].

In principle, any vertex scheduling strategy could be used in the inner-block scheduler. In practice, we prefer low-overhead strategies such as static scheduling and FIFO scheduling to keep the total scheduling overhead small.

4. BLOCK-AWARE EXECUTION ENGINE

We now present the design and implementation of a block-aware execution engine that follows the block-oriented computation model of Section 3. Our engine builds upon GRACE [40], a scalable parallel graph processing engine. In GRACE, users specify the application logic through a vertex update function as described in Section 2. Computation on the graph in GRACE proceeds in iterations, and a subset of all vertices are processed during an iteration. The selection of the subset of the vertices and their order of processing within an iteration depends on a *scheduling policy* that GRACE also enables the user to define. Thus GRACE cleanly separates the application logic (defined through the vertex update function) from the computation strategy (defined through the scheduling policy).

Adapting GRACE to block-oriented computation in a shared-memory parallel environment poses several challenges. First, we want to isolate users from any low-level details of concurrency control inside the engine; users should simply write their application logic through the vertex update function, and the engine should handle all low-level details associated with parallelism. Second, we now have the opportunity to define two different scheduling policies, one at the level of blocks and another at the level of vertices within a block. In the remainder of this section we describe how we addressed these two issues in our execution engine.

4.1 Concurrency Control

As described in Section 3, we partition the input graph into blocks. We use the popular METIS [20] package for this. We split the iteration-based BSP model of GRACE into two levels, which we call *outer iterations* and *inner iterations*. In each outer iteration we process a subset of the blocks; and in each of these blocks we perform one or more inner iterations to execute the vertex update procedure on a subset (or the whole set) of the vertices of the block. In an outer iteration, each thread repeatedly chooses a block of the graph, reads it from shared memory into its local cache, and then performs inner iterations before fetching the next block. The blocks are chosen without replacement, so a block can be processed at most once per outer iteration. Since blocks are generated from highly connected vertices, when the update procedure of a vertex needs to access its neighbor vertices, they are likely to be in the same block and hence already resident in the local cache, so for these vertices the threads can directly read their values without generating cache faults.

However, some of the neighbor vertices will be boundary vertices residing in other blocks (see Section 3.1). Thus simultaneous application of vertex update procedures inside different blocks can access the same vertex data, causing read/write conflicts. This is similar to the read/write conflicts in the vertex-oriented computation model when neighboring vertices are updated simultaneously, and synchronization is required to avoid such conflicts. This synchronization is usually done at the vertex level. However, for computationally light applications, such fine-grained synchronization introduces overhead comparable to the computation logic itself, significantly degrading performance.

For the block-oriented computation model, we can synchronize at the granularity of blocks instead of vertices. One naive approach would be to refrain from scheduling a block if any of its (incoming or outgoing) neighbor blocks is currently being processed. This locking-based scheme guarantees serializability but severely restricts parallelism: two threads cannot concurrently process vertices u and v if they belong to blocks B and B' that are neighbor blocks, even if u and v themselves are not neighbor vertices and so their update procedures could safely be applied in parallel.

To increase parallelism, our block-aware execution engine implements a simple form of multi-version concurrency control that allows neighboring blocks to be processed concurrently within an outer loop with guaranteed snapshot isolation [5]. To update a block B, we create a replica into which the updated block will be written. The thread computing over B writes into the newly created version of B, while it and other threads can read from the old version. In this way, reads and writes from different threads occur in different data versions, so no locking is required. After a thread finishes updating a block, the update is commited. Note that since edge data is read-only, we do not need to maintain multiple versions of the edges within the block. Since each block is processed at most once during each outer iteration, maintaining two versions of the data for each block is sufficient to implement this simple multi-version concurrency control. We maintain a bit for each block indicating which of its versions was most recently updated. When a thread begins processing a newly scheduled block B, it reads and caches the version bits of all the incoming neighbor blocks, and it reads from these versions while processing B. If a neighbor block B' is updated while B is being processed, it does not impact the processing of B because updates to B' are made to the version not being used to process B.

4.2 Scheduling

As GRACE separates the application logic from scheduling policies, it allows users to specify their own execution polices by relaxing data dependencies. The original GRACE system leverages the message passing programming interface to capture data dependencies. To efficiently support our block-aware execution engine with the underlying snapshot-based concurrency control protocol, we replace the GRACE programming interface by a remote read programming interface. Nevertheless, the new GRACE system could also support flexible *vertex-oriented* execution polices with a similar customizable execution interface.

GRACE's original vertex-oriented runtime maintains a dynamic *scheduling priority* on each vertex to support flexible ordering of vertex updates. Users can instantiate a set of runtime functions to define various scheduling policies by relaxing the data dependency implicitly encoded in messages. We adapt the same idea to let users specify their per-vertex scheduling policies in a remote access programming abstraction. Whenever a vertex u finishes its update procedure, the following user-specified function is triggered on each one of its outgoing vertices v:

void OnNbrChange(Edge e, Vertex src, Priority prior)

In this function, users can update vertex v's scheduling priority based on the neighbor's old and new data. For example, to apply Dijkstra's algorithm for the shortest path problem, the vertex's scheduling priority can simply be set to its current tentative distance to the destination, and the above function can be implemented by updating the vertex priority value to the minimum of its current value and the newly updated distance via its changed neighbor:

```
void OnNbrChange(Edge e, Vertex src, Priority p) {
  VtxData vdata = GetNewData(src);
  double newDist = vdata.dist + e.cost;
  if (newDist < GetDstData(e).dist)
    p.Update(newDist, min);
}
```

As in the original GRACE runtime, at the beginning of each iteration the following function is triggered:

void OnPrepare(List<Priority> prior)

in which users can call engine-provided functions to dynamically select the subset of vertices to be updated in the next iteration. For example ScheduleAll can be used to schedule all the vertices that satisfy a user-provided predicate. We refer readers to the original GRACE paper [40] for a detailed description to the functions provided by the engine. As a concrete example, to implement Dijkstra's algorithm, we can choose the single vertex with the smallest priority value in the OnPrepare function. Alternately, since little parallel work is available if we update only one vertex, we could schedule approximately $r \cdot |V|$ of the vertices with smallest priorities by estimating a threshold from a sample of all the priorities and calling ScheduleAll to schedule those vertices with priorities below the estimated threshold. The following code shows the implementation:

```
void OnPrepare(List<Priority> prior) {
  List<Priority> samples = Sample(prior, m);
  Sort(samples, <);
  double threshold = sample[r * m].value;
  ScheduleAll(PriorLessThan(threshold));
}
```

4.2.1 Block-Level Scheduling

By instantiating the above OnNbrChange and OnPrepare functions, users can design flexible per-vertex policies in a remote access programming abstraction. Our block-aware execution engine automatically transforms these policies to follow the block-oriented computation model. To do this, an addition scheduling priority is maintained for each block. Intuitively, a block update should have high priority when some (or most) of its vertices have high priority. Thus we could estimate the block scheduling priority by aggregating the scheduling priorities of vertices in the block. For example, one way of implementing a block-level Dijkstra-like algorithm would define the block priority as the minimum vertex priority. Then, when a block commits the update, the update function adjusts the priorities of its outgoing neighbors as well as its own priority. Algorithm 4 shows a straightforward block update function that maintains block-level priorities. The priorities of vertices in the block are maintained during the block update (line 6), and these are used to calculate the aggregated priority (line 9). When the block is committed, the priorities of outgoing boundary vertices are updated (line 12), and the new priorities are used to update the aggregated priorities of the outgoing blocks (line 15). Here, the function `VertexPriorAggr(B)` calculates the priority of block B from the vertex priorities.

Algorithm 4: Block Update with Priority Maintenance

```
   Input: Block $B$
1  while Inner scheduling queue is not empty do
2      Get a vertex $u$ from inner scheduler ;
3      Reset $u$.prior ;
4      Perform vertex update on $u$ ;
5      foreach $e = (u, v) \in E(B), v \in V(B)$ do
6          OnNbrChange(e, u, v.prior) ;
7      end
8  end
9  $B$.prior = VertexPriorAggr($B$) ;
10 Commit block update ;
11 foreach $e = (u, v) \in E(B), v \in V(B'), B' \neq B$ do
12     OnNbrChange(e, u, v.prior) ;
13 end
14 foreach $B$'s outgoing block $B'$ do
15     $B'$.prior = VertexPriorAggr($B'$) ;
16 end
```

Figure 4 illustrates the block level scheduling of a shortest path computation in a path with four blocks: the upper, left, middle and right block. Each block is a triangle with unit-weight edges, and each vertex is labeled with its current distance estimate. Block updates are scheduled in descending priority value. In the first state shown (Figure 4(a)) the upper block has just been updated, and priorities for the remaining three blocks have been calculated. The middle block has the lowest block priority, so it is scheduled next. When the update to the middle block is committed, the priorities of the neighboring blocks are adjusted, so that the left block now has a lower priority than the right block (Figure 4(b)). The left block is thus processed next, and once the update has been committed, the state is as shown in Figure 4(c).

Because it works at a coarse granularity and ignores potential dependencies among vertices, block-level scheduling is in general less accurate than vertex-level scheduling. However, block-level scheduling results in less scheduling overhead and better cache utilization, and researchers have successfully used block scheduling with block priorities defined by aggregation in several applications [8, 13, 41]. Our framework is general enough to support all the aggregation methods used in these papers. However, for some applications, these aggregates may not be suitable, and users will need to define application-specific block priority functions.

We note two general optimizations to this straightforward implementation. First, many aggregation functions can be maintained incrementally [15]. For example, to maintain the sum of the priorities, we could simply subtract the old priority from the sum and add the new priority into it. In such cases, line 9 and line 15 could be replaced by incremental updates. Second, maintaining vertex priorities in the block after each individual vertex update (line 6) is sometimes unnecessary. If the blocks are run to convergence – a reasonable choice for many applications, as our experiments will show – we do not need to maintain vertex scheduling priorities during block update, which means line 6 could be skipped. Alternatively, if static inner scheduling is used for block updates, the vertex-level scheduling priorities need to be updated only in the last inner iteration. In other words, line 6 would only be executed for the last inner iteration. Both these optimizations are implemented in our system.

As a special case of the second optimization, we observed that for many applications the user-specified OnNbrChange function actually maintains the scheduling priority as an aggregation. For example, for the Dijkstra algorithm, the OnNbrChange function maintains the *MIN* aggregate over the tentative distances. If the same aggregate is used to define block scheduling priority the runtime can skip updating the vertex priority and update the block priority directly when the second optimization is enabled. To implement this, lines 3 , 9 and 15 would be skipped. Line 6 would change to `OnNbrChange(e, u, B.prior)` and line 12 would change to `OnNbrChange(e, u, B'.prior)`. This optimization is supported by the GRACE engine, but must be enabled explicitly.

Using the block priorities, it is straightforward to dynamically select a subset of blocks for each iteration. The engine simply passes the scheduling information to the user-defined OnPrepare to decide the blocks scheduled for the next iteration.

4.2.2 Inner-block Scheduling

As discussed in Section 3, in addition to block-level scheduling we can have low-overhead vertex schedulers within a block. We currently support both static and dynamic inner-block scheduling. To use them, users just need to specify the inner scheduling policy

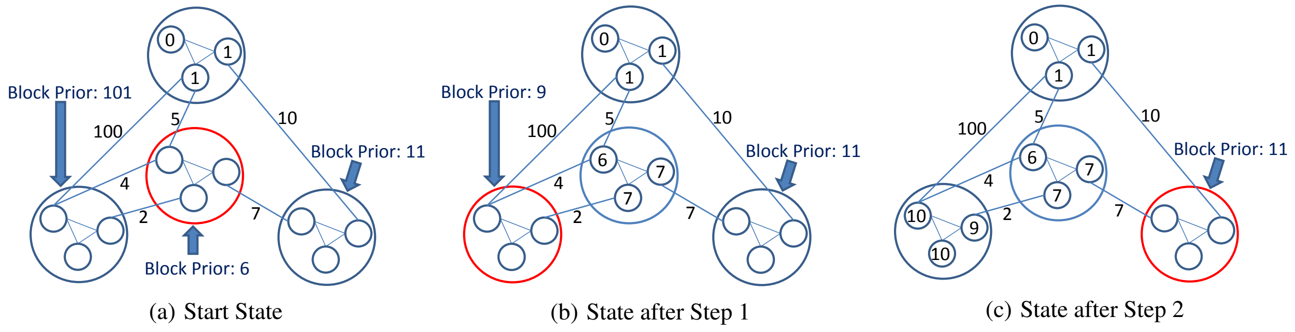

Figure 4: Illustration for Block Level Scheduling

and the parameters.

Inside-Block Ordering. Inside a block, vertex updates are done sequentially. Since only one assigned thread is responsible for updating the block, there are no update conflicts inside the block. As each update is immediately visible to later updates, the update ordering for each inner iteration can make a significant difference. Currently, we provide two pre-defined static inner schedulers: fixed-order sweeping and alternating sweeping. For fixed-ordering sweeping, the engine updates vertices in the same order in each inner iteration, while alternating sweeping reverses the order between inner iterations.

Dynamic Inner Scheduling. Instead of statically sweeping over the vertices inside the block for a fixed number of sweeps or until convergence, we may use dynamic inner scheduling, potentially eliminating update function calls for vertices whose neighbors have unchanged data. To this end, we have implemented a low-overhead dynamic inner scheduler. Each block has a queue of vertices whose neighbor data has changed since the last update. Every incoming boundary vertex also has an extra outer scheduling bit. For a given block B and incoming boundary vertex v, this outer scheduling bit is set if the data on any of v's incoming neighbors outside block B have changed.

When block B is chosen to be updated, all the vertices with outer scheduling bits set are added to the queue. The GRACE engine then repeatedly pops a vertex u from the queue and invokes the update function on it. If the vertex data of u changes, the engine pushes all of u's outgoing vertices inside the block B onto the queue. The block update continues until the queue is empty, or a pre-defined maximum number of updates is reached. In the latter case the vertices remaining in the queue wait for the next time that block B is chosen to be updated.

Once block B commits, the GRACE engine iterates over all outgoing boundary vertices u that have changed, setting the outer scheduling bit for all vertices v s.t. $(u, v) \in E, v \notin B$.

Maximum Number of Inner Iterations. We need to decide the maximum number of inner iterations per block update. At one extreme, we can perform only a single iteration; i.e., update each vertex once per block update. This is similar to the traditional vertex execution model, except it yields better cache performance because we usually read the same vertex data repeatedly when updating a block. At the other extreme, we can set this number to infinity, so that each block is repeatedly updated until it converges for its current boundary values. How to make this choice depends on the ratio of data access to computation cost and on how much additional inner iterations benefit convergence. If little computation is done for each byte fetched from memory and each inner iteration significantly accelerates global convergence, the number of inner iterations should be large. On the other hand, if the computation is heavy, or if additional inner iterations do not accelerate global convergence very much, the maximum number of iterations should be small.

Given the fixed boundary data $S_{NV(B)}$ and $S_{NE(B)}$, it is possible for a block to converge before the maximum number of inner iterations is reached. In this case the block update can be terminated. This situation is likely when the global graph data is close to convergence.

5. EXPERIMENTS

We added block updates to GRACE, a shared-memory parallel graph processing framework implemented in C++ with pthreads [40]. We did not change its vertex-oriented programming model, but modified the runtime as described in Section 4 to support block-aware execution. Although this preliminary implementation is not (yet) a general engine – currently the block level schedulers are manually coded, based on the scheduling policies used in the experiments – it can already demonstrate the key techniques described in Section 4 and hence be used to validate their performance benefits.

The original GRACE runtime provides two dynamic scheduling policies: Eager and Prior. Users need only provide an application-specific function to compute priorities. Both policies schedule only a subset of the vertices in each tick: for the Eager policy, a vertex is scheduled if a neighbor's data has changed; and for the Prior policy, only the $r \cdot |V|$ highest-priority vertices are scheduled for update, where r is a configurable selection ratio. Both of them are extended to be executed with the block-oriented computation model as we discussed in Section 4.

Our experimental evaluation has three goals. First, we want to verify that our block-oriented computation model can improve end-to-end performance compared with the vertex-oriented computation model. Second, we want to show that this end-to-end performance gain comes from both better cache behavior and lower scheduling overhead. Last, we want to evaluate the effect of inner-block scheduling policies on the convergence rate.

5.1 Applications

Personalized PageRank. Our first application, Personalized PageRank (PPR), is a PageRank computation augmented with a personal preference vector [17]. We randomly generated a sparse personalization vector in which all but 1% of the entries are zeros. The standard iterative algorithm is a Richardson iteration for solving a linear system. The natural algorithm in GRACE is a Jacobi or Gauss-Seidel iteration, while the natural approach in our block-aware execution engine is a block Jacobi or block Gauss-Seidel

Table 2: Dataset Summary

Data Set	Vertices $\times 10^3$	Edges $\times 10^3$	Partition Time (s)	Application
DBLP	968	7,050	38	PPR
Web-Google	876	5,105	34	PPR
LiveJournal	4,848	68,994	659	SSSP
3D Grid	1,728	9,858	N/A	Etch Sim
UK02	18,520	298,114	1034	PPR

iteration [11, 4]. We declare convergence when all the vertex updates in an iteration are less than a tolerance of 10^{-3}.

Single-Source Shortest Path. Our second application is Single-Source Shortest Paths (SSSP), where each vertex repeatedly updates its own distance based on the neighbors' distances from the source. GRACE's original vertex-oriented execution with eager scheduling corresponds to the Bellman-Ford algorithm, and its prioritized scheduling corresponds to Dijkstra's algorithm. We are not aware of any algorithms in the literature that correspond to approaches in our block-aware excution engine, though there has been work on similar blocked algorithms for the related problem of solving the eikonal equation [8]. All the variants of this algorithm converge exactly after finitely many steps, and we declare convergence when no vertex is updated in an iteration.

Etch Simulation. Our third application is a three-dimensional Etch Simulation (Etch Sim) based on an eikonal equation model [33]. The simulation domain is discretized into a 3D grid and represented as a graph. The time at which an etch front passes a vertex can be computed iteratively based on when the front reaches its neighbors. The vertex-oriented execution engine with eager scheduling in GRACE corresponds to the Fast Sweeping method for solving the equations, while its original prioritized scheduling corresponds to the Fast Marching method. Our block-aware execution engine with block-level eager scheduling corresponds to the Fast Sweeping method with Domain Decomposition [34], while our block-aware execution engine with block-level prioritized scheduling corresponds to the Heap-Cell method [8]. As with SSSP, all these algorithmic variants converge exactly after finitely many steps, and we declare convergence when no vertex is updated in an iteration.

5.2 Experimental Setup

Machine. We ran all our experiments using an 8-core computer with 2 Intel Xeon L5335 quad-core processors and 32GB RAM.

Datasets. Table 2 summarizes the datasets we used for our applications. For PPR, we used a coauthor graph from DBLP collected in Oct 2011, which has about 1 million vertices and 7 million edges. We also used a web graph released by Google, which contains about 880,000 vertices and 5 million edges. We omit the running time results on DBLP graph due to the space limitations. For Shortest Path, we used a social graph from LiveJournal with about 5 million vertices and 70 million edges. For the Etch Simulation application, we constructed a 3D grid that has $120 \times 120 \times 120$ vertices. Finally, we demonstrate the performance of our system on a larger example, a web graph of the `.uk` domain crawled in 2002. This graph contains about 18 million vertices and 300 million edges. Vertices are ordered randomly in all the datasets except the 3D grid dataset.

Partition Time. For the DBLP, Google, and LiveJournal graphs, we used METIS [20] to partition the graph into blocks of around 100. For the `.uk` web graph, we partitioned into blocks of size 400. We report the partitioning times in Table 2. While partitioning is itself expensive, our focus is problems where the partitioning work will be amortized over many executions of the main computation. Given that relatively small blocks appear useful in practice, recently-developed fast algorithms for bottom-up graph clustering [35] may perform better on these problems.

Scheduling. As mentioned in Section 4, static scheduling and two common dynamic scheduling policies (Eager and Prior) are implemented in the GRACE runtime. To use Prior scheduling, users must also provide the application-specific priority calculation. For the Eager scheduling policy, the scheduling priority for a vertex is a boolean value indicating whether any neighboring vertex data has changed. Thus we use boolean *OR* as the block priority aggregation, which means a block would be scheduled if its boundary data has changed, or its last update did not run to convergence. For the Prior scheduling policy, each vertex holds a float value to indicate its priority. The priority aggregation used for SSSP and EtchSim is *MIN* in our experiments, while the aggregation used for PPR is *SUM*. Notice that since we use *MIN* for SSSP, it is eligible for the direct block-priority update optimization described in Section 4.

5.3 Results

5.3.1 Block Size

The Etch Simulation application has a natural grid structure, and we used sub-grids of size $b \times b \times b$ as blocks, while for other applications we used METIS to partition the graph into blocks of roughly equal size. The best block size depends on many factors, including characteristics of the machine, characteristics of the data and application, how the graph is partitioned, and how block updates are scheduled. In our applications, the performance was only moderately sensitive to the block size, as we illustrate in the first column of Figure 5. Because block sizes between 100 and 400 performed well for these examples, we chose a default block size of 100 for the PPR and SSSP test cases except the UK dataset, and used a $5 \times 5 \times 5$ sub-grid as a block for the Etch Simulation. For the much larger UK dataset, we set the block size to be 400.

5.3.2 End-to-End Performance

To see how the block-oriented computation model performs compared to the vertex-oriented computation model, we ran each of our example applications under different scheduling polices. The mean run times for the scheduling policies are shown in the second column of Figure 5; the bars labeled Vertex and BlockCvg correspond respectively to applying this schedule to individual vertex updates and to block updates. Here when a block is scheduled, each vertex is repeatedly updated until the block data converges. We omit the time for the static scheduling policy for the Etch Simulation application from Figure 5. While a well-chosen static schedule leads to very fast convergence for this problem [43], the naive static schedule in our current implementation takes much longer than the two dynamic scheduling polices: 15.9s for the vertex model and about 3.6s for the block model.

In our experiments, the best scheduling policy under the vertex model is also the best scheduling policy under the block model. More specifically, for the SSSP and Etch Simulation applications, the best scheduling policy is the prioritized policy. For the PPR application, the best scheduling policy depends on the dataset characteristics. On the Google and UK graph, dynamic scheduling performs better than static scheduling, while on the DBLP graph, the static scheduling policy performs best. This is because the DBLP graph has a much larger clustering coefficient than the Google and

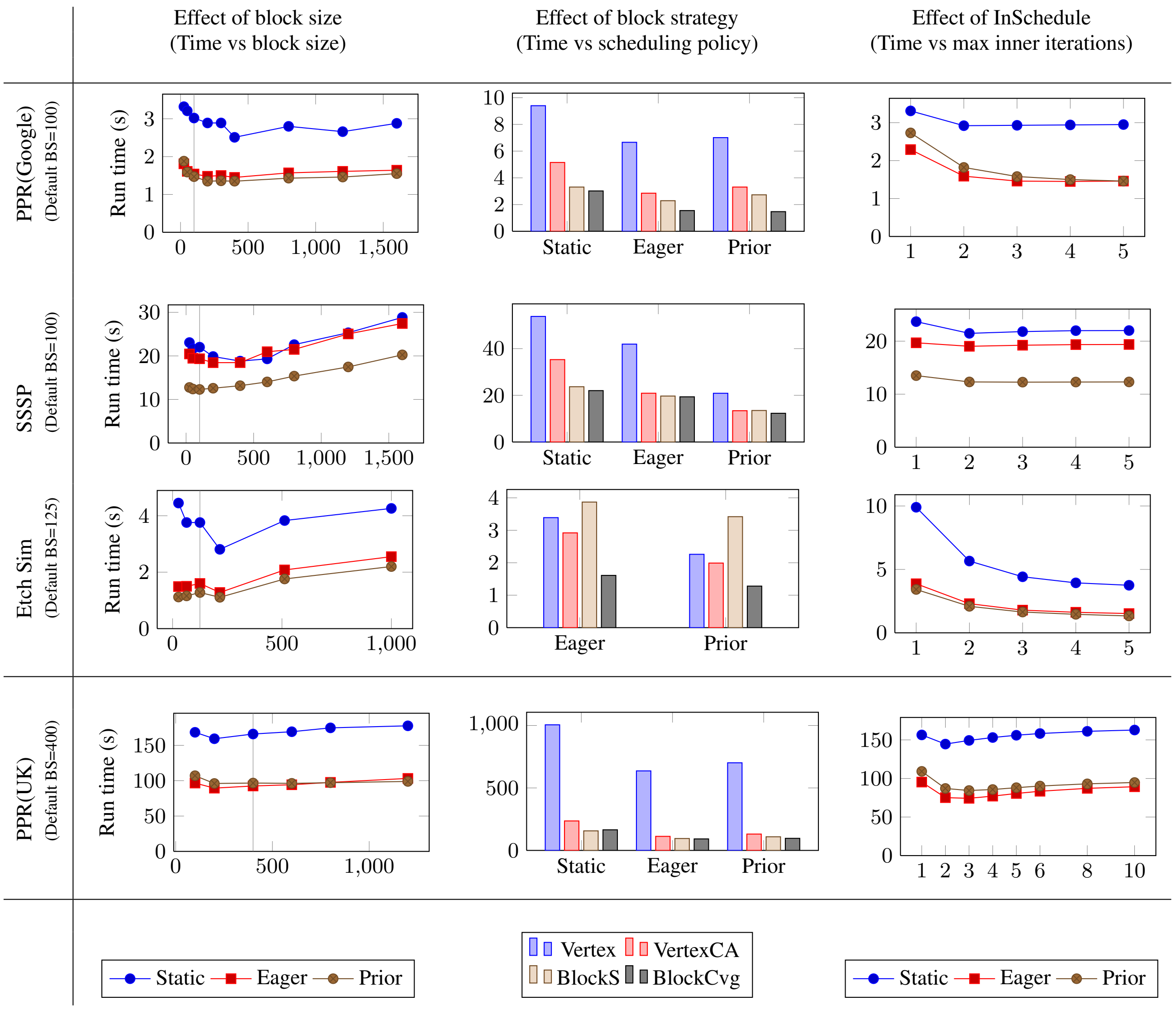

Figure 5: Timing details for five application scenarios.

UK graph. Thus the computational savings due to dynamic scheduling polices are smaller and are outweighed by the high overhead of the dynamic schedulers themselves. Moreover, dynamic scheduling policies tend to schedule vertices with higher degrees in the PPR application, which makes the vertex updates more expensive.

In general, the blocked computation outperforms the corresponding vertex-centric computation under each scheduling policy. In the PPR application, our block engine runs $3.5\times$ – $7.0\times$ faster than the vertex-centric computation for the best scheduling policy. For the SSSP and Etch Simulation applications, we cut the run time roughly in half. Also, we observed that the block-oriented computation model is more robust to the "wrong choice" of scheduling policy. For example, the vertex-centric prioritized scheduler is about $2.5\times$ slower than the vertex-centric static scheduler, but the block prioritized scheduler is about 60% slower than the block static scheduler. This is because by scheduling blocks rather than vertices, we significantly reduce the total scheduling overhead.

5.3.3 *Analysis of Block Processing Strategies*

Recall that the block-oriented computation model has three main benefits: (1) It has a better memory access pattern due to visiting vertices in the same block together. (2) It has reduced overhead for isolation due to providing consistent snapshots at block level instead of at vertex level. (3) It can achieve better cache performance by doing multiple iterations in a block. To understand how each of these benefits contributes to improved end-to-end performance, we analyzed the run time for each application in two execution models that are hybrids of the pure vertex-oriented computation model and the pure block-oriented computation model used in general performance comparison. We show the running times of all these schedulers in the second column of Figure 5.

To understand how the memory access pattern affects performance, we used the cache aware vertex-oriented computation model (VertexCA) introduced in Section 2. Recall this execution model still updates one vertex at a time, and makes scheduling decisions at the vertex level. However, it is aware of the graph partitioning and updates the vertices in block order; i.e., it updates all the scheduled vertices of a given block before proceeding to the next block. By doing so it achieves better temporal locality. We also report the running time for two different inner-block schedulers: the simple

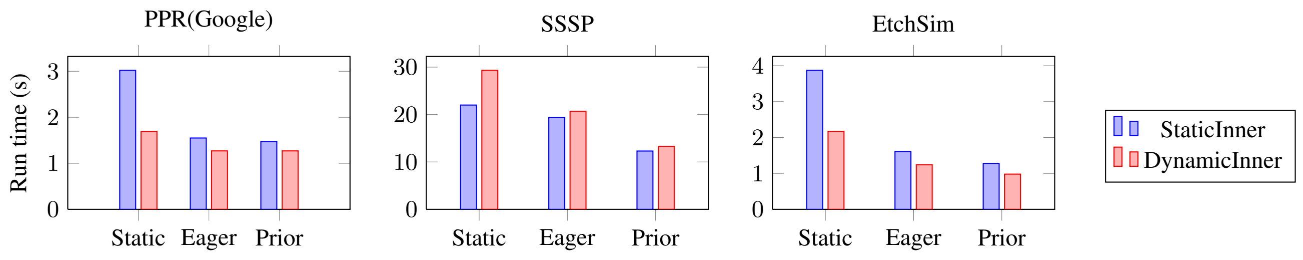

Figure 6: Effect of dynamic inner scheduling with different block level scheduling policies.

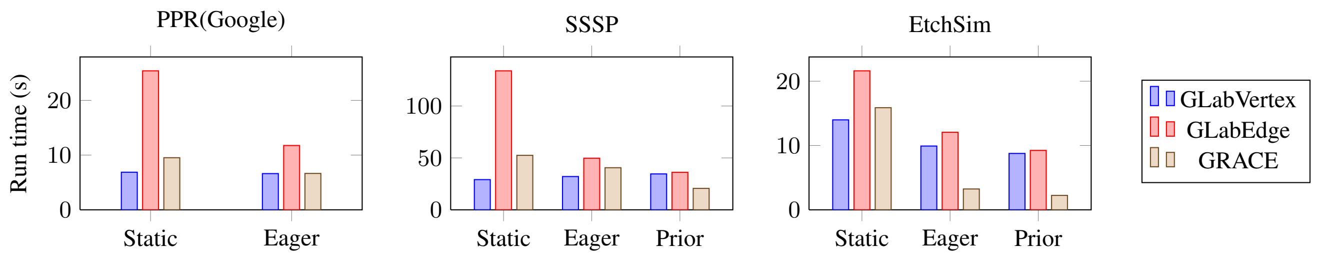

Figure 7: Effect of scheduling policies in GraphLab (with vertex or edge consistency) and in GRACE.

block model, which just sweeps all the vertices once (BlockS), and the convergent block model, which iteratively updates the block data until it converges (BlockCvg). Note that the major difference between the simple block model and the cache-aware vertex model is the isolation and scheduling overhead. GRACE uses snapshot isolation; thus, before updating a vertex or block the engine is responsible for choosing the right version of the data to be passed to the update function. By making this decision at the block level rather than at the vertex level, the average overhead is greatly reduced. Finally, the difference between the simple block model and the convergent block model is that the CPU is able to do more work on data residing in the cache when the memory is saturated.

With the cache-aware vertex model, we observed a significant reduction in running time of the PPR and SSSP applications, achieving savings of 36% to 52% on the best scheduling policy except for the UK graph. We observed much more saving on the larger UK graph, in which the cache-aware vertex model is more than 5 times faster than the vertex model. In contrast, the cache-aware vertex model only saves about 10% of the time for the prioritized policy in Etch Simulation application. This is because the grid-structure graph used in the Etch Simulation already has a regular memory access pattern, while the memory access pattern for arbitrary graphs could be quite random.

Switching from the cache-aware vertex model to the convergent block model, we save about half the time for PPR on the DBLP and Google graphs. We see similar savings for the Etch Simulation application, but for different reasons. For PPR, performance improves from the cache-aware vertex model to the simple block model and finally to the convergent block model. However, for Etch Simulation, the simple block model has worse performance than the cache-aware vertex model, while the convergent block model has much better performance. This is because scheduling at the block level wastes many vertex updates in this case. For PPR on the UK graph, switching from VertexCA to BlockCvg reduces the running time by about 20% for the eager scheduling policy, which is not as significant as PPR on other two datasets. This is because running until convergence inside a block gives only a slight improvement. As we will see in Section 5.3.4, an inexact block solve improves the overall running time on this dataset. For SSSP, BlockS has roughly the same running time as VertexCA, while BlockCvg only improves the performance by 10%. This is because social networks obeying a power-law are hard to partition – in the partitions we used in our experiments, more than half the edges are cutting edges. Thus updating the block until convergence contributes little to achieving global convergence. Researchers have shown that overlapping partitions would help this problem [1], and some recent emergent graph processing frameworks such as PowerGraph [12] have designed their programming interfaces to naturally support computation on overlapping partitions. We expect this block computation model would have more benefits on social graph computations for these frameworks. We oalso bserved that the direct block-priority update optimization to BlockCvg reduces the running time from 12.3 seconds to 10.5 seconds, and makes BlockCvg 22% faster than VertexCA.

5.3.4 *Effect of Inner-Block Scheduling*

In this subsection we focus on the effect of inner-block scheduling. In particular, we have seen that updating each vertex multiple times in a single block update often improves performance. For example, if the boundary data of the block has already converged, then iterating over the vertices until the block data converges is a natural way to define the block update function. However, this could be a poor choice if the boundary data of the block is incorrect. The tradeoff is that doing more updates inside the block leads to better cache performance, but it could waste CPU time if the boundary data has not converged. As we mentioned in Section 4, we can set a maximum number of inner iterations I_θ, and terminate the block update after I_θ sweeps even if the block has not yet converged.

To understand this tradeoff, we plot running time against the number of inner-block iterations I_θ in the third column of Figure 5. For PPR, the best performance occurs around $I_\theta = 3$, and after that the running time remains the same as running until convergence for both DBLP and Google datasets. However, for the UK dataset, further increasing I_θ increases the overall run time significantly. Specifically, running until convergence is 24% slower than setting $I_\theta = 3$. On the other hand, for the Etch Simulation application, more inner iterations always yields better performance. We believe that besides the application characteristics, the higher diameter of the graph also favors more inner iterations because they help information propagate across the graph faster.

Dynamic scheduling may also be used inside blocks to reduce the number of updates, at the cost of paying some extra scheduling overhead. To study this tradeoff, we compare the run times for static and dynamic inner scheduling in Figure 6. For all three applications, dynamic inner scheduling reduces the number of vertex updates; nonetheless, static scheduling outperforms dynamic scheduling for the SSSP problem, because the computational saving is outweighed by the scheduling overhead. For PPR, we observed that dynamic inner scheduling is slightly faster than the static inner scheduling on Google graph, while it is slower than the static inner scheduling on the DBLP graph. However, dynamic inner scheduling yields nearly 25% improvement in the Etch Simulation application, as the vertex update function is slightly computationally heavier than the previous two applications and the convergence for this problem is particularly sensitive to update order.

5.3.5 *Comparison with GraphLab*

To evaluate the performance of the vertex execution model implemented in GRACE, we compare the running time of GRACE with the GraphLab shared-memory multicore version on all three applications under different scheduling policies. Recall that GraphLab provides two different isolation levels for concurrent vertex updates: vertex consistency, which allows two neighboring vertices to update simultaneously, and edge consistency, which guarantees serializability of updates. As pointed out by the GraphLab authors, for some graph applications vertex consistency can produce inaccurate results, as it does not avoid read/write conflicts on neighboring vertices [25, 26]. We report GraphLab's run time under both isolation levels in Figure 7.

For Eager and Prior scheduling, GRACE's run time is between that of GraphLab with vertex consistency and GraphLab with edge consistency. This is because the isolation level used by GRACE – snapshot isolation – is between GraphLab's vertex consistency and GraphLab's edge consistency. The only exception is the Eager scheduling for eikonal equation, in which GRACE is faster than GraphLab whether vertex or edge consistency is used. This is because the corresponding scheduler in GraphLab executes more than twice as many updates as GRACE.

For Prior scheduling, GRACE is always faster than GraphLab, because it makes the scheduling decision by iterations rather than by vertex [40]. Thus we expect the block model could benefit GraphLab even more than GRACE, since GraphLab has higher scheduling overhead.

6. RELATED WORK

Most existing graph programming frameworks can be categorized into two groups. The first group are mainly designed for distributed memory environments and are based on the Bulk Synchronous Parallel (BSP) [38] model, which allows processors to compute independently within each iteration and uses global synchronization between iterations for processors to communicate. Example frameworks in this group include PEGASUS [19], Pregel [27], PrIter [42], AsyncMR [18] and Naiad [28], which are mostly built on MapReduce or DryadLINQ with a higher level programming model. The second group of frameworks are mainly designed for multi-core shared memory architectures and usually do not apply global synchronization barriers for threads to communicate. Instead, threads can proceed asynchronously, which enables various scheduling of computation tasks to achieve better convergence. Consistency is guaranteed either by a fine-grained locking mechanism to synchronize shared data access or by requiring all operations to be commutative and able to be rolled back in case of a data race. Frameworks in this group include GRACE [40], GraphLab [25], GraphChi [23] and Galois [22]. One note here is that GraphChi is designed to store graphs out-of-core and its "sliding window" idea bears some resemblance to our cache-aware approaches.

Nevertheless, both these groups provide a local vertex computational interface to users to code their application logic, and both groups execute the coded applications in a per-vertex update manner. As we have illustrated in Section 1, such a per-vertex update model, while appropriate for programming, is not an ideal execution model for computationally light applications due to its poor locality and high demand for memory bandwidth.

The idea of having more local computations to reduce the communication overhead has also been suggested in the AsyncMR framework [18]. However, since the AsyncMR framework follows the MapReduce model, the main communication overhead is global synchronization. We are applying this optimization technology to GRACE, an asynchronous framework in which global synchronization is not the main communication overhead. In addition, because of its underlying MapReduce framework, AsyncMR does not support flexible dynamic outer scheduling.

Blocking is widely used in high-performance computing to improve memory access patterns. A textbook example is blocked matrix multiplication [31, Section 5.3], which is the basic building block for high-performance dense linear algebra libraries like LAPACK [2]. For large problems, such cache-blocked dense linear algebra codes typically use $O(\sqrt{M})$ floating point operations per cache miss, where M is the cache size [10]. Similar optimizations apply to graph algorithms such as Floyd-Warshall that are structurally similar to dense linear algebra operations [30]. In contrast, sparse matrix algorithms have relatively irregular memory access patterns, and it is more difficult to block them for efficient cache use. Recent research addresses this to some extent by automatically reorganizing the graph data structures used to store sparse matrices in order to optimize key operations such as sparse matrix-vector multiplication [16, 39, 29]. Blocking is also used for improved locality and convergence in many iterative solvers for linear and nonlinear equations and optimization problems; examples include block Jacobi and block Gauss-Seidel methods for accelerating iterative solvers [4], domain decomposition and substructuring methods used in linear and nonlinear PDE solvers [34], and block coordinate descent methods in optimization [6], etc. Because the local block updates are relatively expensive, these methods often achieve good communication-to-computation ratios [14].

7. CONCLUSIONS

In this paper we presented a new block-oriented computation model that is compatible with vertex-centric programming abstraction but executes a block of highly connected vertices at a time instead of one vertex at a time. Our block-aware graph execution engine can achieve better cache performance and enables more flexible block-level and vertex-level scheduling to further accelerate convergence. In particular, it can provide near-interactive runtime and better multicore speedup than current per-vertex computation models for a large class of graph processing applications.

We believe that this work is only a first step towards more confluence between the HPC and the database communities and a major step towards enabling iterative graph processing with interactive response times, a fascinating topic for future research. We are aware that other framework providers are actively improving the performance of their graph processing engines. However, our block-oriented execution model provides an orthogonal perspective on optimizing computationally light graph applications, and we believe that it is applicable to other frameworks as well.

This research has been supported by the National Science Foundation under Grants IIS-0911036 and IIS-1012593, by the Air Force Office of Scientific Research under Award FA9550-10-1-0202, the iAd Project funded by the Research Council of Norway, and by gifts from NEC and Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

8. REFERENCES

[1] R. Andersen, D. F. Gleich, and V. S. Mirrokni. Overlapping clusters for distributed computation. In *WSDM*, 2012.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] S. T. Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *SC*, page 27, 1995.

[4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[5] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD*, 1995.

[6] D. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1995.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[8] A. Chacon and A. Vladimirsky. Fast two-scale methods for eikonal equations. *SIAM J. Scientific Computing*, 34(2), 2012.

[9] D. Crandall, A. Owens, N. Snavely, and D. P. Huttenlocher. Discrete-Continuous optimization for large-scale structure from motion. In *CVPR*, pages 3001–3008, 2011.

[10] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[11] G. H. Golub and C. F. Van Loan. *Matrix computations (3. ed.)*. Johns Hopkins University Press, 1996.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[13] M. Griebel and P. Oswald. Greedy and randomized versions of the multiplicative Schwarz method. *Linear Algebra Appl.*, 437(7):1596–1610, 2012.

[14] W. D. Gropp and D. E. Keyes. Domain decomposition on parallel computers. *Impact Comput. Sci. Eng.*, 1:421–439, 1989.

[15] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *EDBT*, 1996.

[16] E.-J. Im, K. Yelick, and R. Vuduc. SPARSITY: An optimizing framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18:135–158, 2004.

[17] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, 2003.

[18] K. Kambatla, N. Rapolu, S. Jagannathan, and A. Grama. Asynchronous algorithms in MapReduce. In *CLUSTER*, 2010.

[19] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, 2009.

[20] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *DAC*, 1997.

[21] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[22] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[23] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[24] C.-K. Liang, C.-C. Cheng, Y.-C. Lai, L.-G. Chen, and H. H. Chen. Hardware-efficient belief propagation. In *CVPR*, 2009.

[25] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, 2010.

[26] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.

[27] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[28] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[29] R. Nishtala, R. Vuduc, J. Demmel, and K. Yelick. When cache blocking sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communications, and Computing*, 18:297–311, 2007.

[30] J.-S. Park, M. Penner, and V. Prasanna. Optimizing graph algorithms for improved cache performance. In *IPDPS*, 2002.

[31] D. Patterson and J. Hennessey. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[32] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76:036106, 2002.

[33] J. A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, 1999.

[34] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Difference Equations*. Cambridge University Press, 1996.

[35] D. Spielman and S.-H. Teng. A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning. *SIAM J. Comput.*, 42(1):1–26, 2013.

[36] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, pages 1222–1230, 2012.

[37] P. Stutz, A. Bernstein, and W. W. Cohen. Signal/Collect: Graph algorithms for the (semantic) web. In *ISWC*, 2010.

[38] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), 1990.

[39] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, 2005.

[40] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous large-scale processing made easy. In *CIDR*, 2013.

[41] D. Wingate, N. Powell, Q. Snell, and K. Seppi. Prioritized multiplicative Schwarz procedures for solving linear systems. In *IPDPS*, 2005.

[42] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: A distributed framework for prioritized iterative computations. In *SOCC*, 2011.

[43] H. Zhao. Parallel implementation of fast sweeping method. *Journal of Computational Mathematics*, 25(4):421–429, 2007.