Constrained Data-Driven Parallelism

Tim Harris Yossi Lev Victor Luchangco
Virendra J. Marathe Mark Moir

Oracle Labs

{timothy.l.harris,yossi.lev,victor.luchangco,virendra.marathe,mark.moir}@oracle.com

Abstract

In data-driven parallelism, changes to data spawn new tasks, which may change more data, spawning yet more tasks. Computation propagates until no further changes occur. Benefits include increasing opportunities for fine-grained parallelism, avoiding redundant work, and supporting incremental computations on large data sets. Nonetheless, data-driven parallelism can be problematic. For example, convergence times of data-driven single-source shortest paths algorithms can vary by two orders of magnitude depending on task execution order. We propose *constrained* data-driven parallelism, in which programmers can impose ordering constraints on tasks. In particular, we propose new abstractions for defining groups of tasks and constraining the execution order of tasks within each group. We sketch an initial implementation and present preliminary performance results suggesting that our approach enables new efficient data-driven implementations of a variety of graph algorithms.

1. Introduction

Exploiting multicore systems requires parallel computation, preferably with minimal synchronization. A promising approach is *data-driven parallelism*, in which computation is broken into *tasks*, each of which must run when some data is modified. Such tasks may modify additional data, thereby triggering additional tasks. Thus, changes to data "drive" the parallel computation.

Consider, for example, the *single-source shortest paths (SSSP) problem*: given a weighted graph with no negative-weight cycles, compute the length of the shortest paths to each node from a single source node. We can solve SSSP by assigning the source a distance estimate of 0 and every other node a distance estimate of ∞, and then performing *relaxation steps* on the graph's edges until no more are possible. Each relaxation step considers an edge from x to y, and whether going to x and then following this edge provides a shorter path than the current distance estimate to y.

This computation is well-suited for parallelism because relaxation steps can run in any order, and synchronize only when their edges share a node. It is well-suited to being data-driven because relaxation steps must be performed initially only on edges of the source node, and thereafter, only on edges of nodes whose distance estimates have been updated.

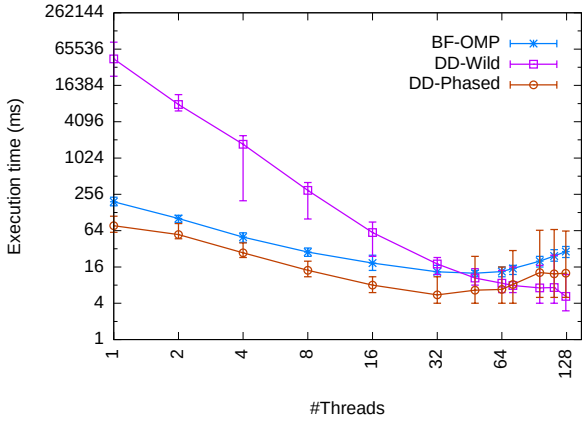

Figure 1: Time to compute SSSP with OpenMP (BF-OMP) and data-driven algorithms (DD-Wild, and DD-Phased).

Although this computation is correct regardless of the order in which edges are relaxed, this order can have a profound impact on execution time: the best order (such as the one used by the Dijkstra algorithm) requires one relaxation step per edge; a bad order can induce an exponential number of relaxation steps. As Figure 1 illustrates, this problem is not merely theoretical. The figure shows three variants of SSSP running on the ca-HepPh graph from SNAP [23]. BF-OMP is a parallel version of Bellman-Ford: computation proceeds in rounds, and every edge is relaxed once per round using an OpenMP parallel for loop. DD-Wild is data-driven: an edge is relaxed when it is incident on a node whose distance estimate has changed. This tends to produce a depth-first traversal of the graph, which is a bad order for computing SSSP. Thus, although it scales well, with one thread, DD-Wild is over 200 times slower than BF-OMP. DD-Phased—a data-driven implementation that uses the abstractions introduced in Section 2 to constrain task execution order—outperforms BF-OMP at every thread count. We discuss these results in more detail in Section 4.

We argue that data-driven parallelism benefits significantly from the ability to constrain the order in which tasks run. Furthermore, we argue that choosing the ap-

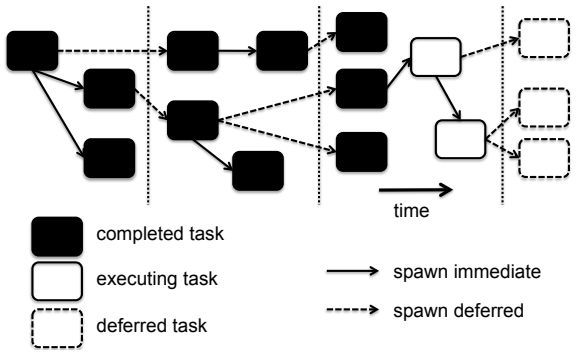

Figure 2: Snapshot of a CDDP computation: Tasks spawn other tasks, either immediately or deferred. Deferred tasks are not executed until all other tasks complete, implicitly partitioning tasks into phases.

propriate constraints is a property of an algorithm, and so should be expressed in the source code (rather than by picking a plug-in scheduler for a whole application).

Section 3 describes our prototype. Section 4 shows the power of constraining data-driven parallelism via our abstractions in various ways using several benchmarks. Section 5 discusses related work. Section 6 concludes.

2. Programming Model

Computation is divided into *tasks* executed by *threads*. Each task belongs to a *task group*. A task may *spawn* a new task via a *parallel for loop* or a *trigger*, which specifies data-driven computation:

```
x triggers [deferred] f(void *d);
*y triggers [deferred] g(void *d);
```

We say x is a *direct trigger* for f and y an *indirect trigger* for g. A trigger is a *deferred trigger* if it is declared with the optional deferred keyword; otherwise, it is an *immediate trigger*. When such declarations are in effect, writing to x spawns a task to run f, and modifying data by dereferencing y spawns a task that runs g. (See Section 4.1 for an example.) The functions f and g take a single argument, which is a pointer to the data that was modified. This has been sufficient for all use cases we have explored so far, though additional arguments may be useful in the future.

A task spawned by a deferred trigger is *deferred*: it is not executed until all nondeferred tasks in its task group have completed. This implicitly partitions the tasks in a task group into a totally ordered series of *phases*: an immediate trigger spawns tasks in the current phase, a deferred trigger spawns tasks in the next phase, and no task is executed until all tasks in previous phases have completed (see Figure 2). Thus, at any time, every unfinished task is in either the current phase or the next phase of its task group.

A *task group*, declared as follows, is a set of tasks.

```
taskgroup {
    // code
} t;
```

When this block is executed, a new task group t is created. Initially, a new task group has a single task, running the code in the declared block (executed by the thread that entered the block). A newly spawned task belongs to the same task group as the task that spawned it. In our current model, every task belongs to exactly one task group. There is a single anonymous task group to which all initial tasks belong.

A task group's only method—WaitForGroup()—blocks until every task in that task group has completed. Note that deadlock cannot arise solely from calls to WaitForGroup because the name of a task group is not in scope in the code block and we provide no mechanism to pass task groups (i.e., there is no named type for task groups).

3. Implementation Overview

To evaluate constrained data-driven parallelism (CDDP), we have implemented a simple prototype using C++ macros and a runtime library. We support triggering via explicit calls to runtime library functions (one for immediate triggering, and one for deferred). These functions take as arguments the object that triggered the call and the handler function to run. Our runtime system manages tasks using a work-stealing scheduler based on the Chase-Lev deque [4].

We implement task groups with a TaskGroup class, exposing BeginTaskGroup and EndTaskGroup methods. BeginTaskGroup begins the "scope" of a task group which owns all tasks spawned until the matching call to EndTaskGroup. EndTaskGroup returns a pointer to the task group object, which can be passed to WaitForGroup.

Each task group tracks whether any tasks remain to be executed in the current and next phases using SNZI objects [7, 14]. Each thread keeps the deferred tasks that it spawns in a thread-local deferred-task bag. When no tasks remain in the current phase of a task group, each thread moves any deferred tasks it has from the bag of that task group to its deque. When no thread has any deferred task for a task group, the task group's execution is compete.

4. Evaluation

We evaluate CDDP using SSSP, Communities (a graph clustering algorithm [20]), and BC (betweenness centrality, a social network analysis algorithm [15]). We compare our data-driven solutions against parallel implementations using OpenMP [17]. CDDP delivers competitive, or significantly better, performance across most benchmarks and inputs.

We also ported the discrete-event simulator from the LoneStar benchmark suite [12], in which the computation is structured entirely via data dependencies between tasks. The performance of our CDDP solution is consistent with results reported by Kulkarni et al. [12] (we omit full details for brevity).

Experiments were run on an Oracle T5140 series machine, comprising two 1.2 GHz Niagara T2 chips with a total of 128 hardware thread contexts (8 cores per chip, 8 hardware threads per core). Each chip has an 4MB on-board L2 cache, and each core has a 8KB L1 data cache shared between its threads. We use the Oracle Solaris Studio 12.1 C++ compiler at optimization level xO5.

4.1 Single-Source Shortest Paths (SSSP)

As discussed in the introduction, relaxation algorithms such as SSSP are well-suited for data-driven parallelization. To reduce the overhead of spawning tasks, we spawn a single task that relaxes all the edges of a node whose distance estimate is updated, rather than spawning a separate task for each edge. We also introduce a per-node pendingRelaxation flag that indicates whether the node's distance estimate has been modified since its edges were most recently relaxed, and trigger relaxation when this flag is set to **true**, rather than when the distance estimate is updated. This allows multiple successive updates to be handled by a single task that calls the following function:

```
1   void RelaxNeighbors(Node *n)
2     n->pendingRelaxation = false
3     forall (Node *k in neighbors(n))
4       newDist = n->dist + weight(n,k)
5       bool* pendingFlagP = &k->pendingRelaxation
6       *pendingFlagP triggers RelaxNeighbors(k)
7       currDist = k->dist
8       while (newDist < currDist)
9         if (CAS(&k->dist, currDist, newDist))
10          // Triggers on successful CAS
11          CAS(pendingFlagP, false, true)
12          break
13        currDist = k->dist
```

Note the use of an indirect trigger to trigger RelaxNeighbors when the flag changes from **false** to **true** at line 11, but not when it is set to **false** at line 2. We use CAS at line 11 so that RelaxNeighbors is triggered only when the flag changes (i.e., when the CAS is successful).

We experimented with three versions of this algorithm. DD-Wild uses "classical" work-stealing, in which each thread pushes and pops tasks on one end of its deque, and thieves steal from the other end. With only a single worker thread, this implies LIFO execution of tasks. DD-Phased uses the same algorithm and runtime, but with deferred triggering (adding the deferred keyword at line 6). Finally, DD-Fifo uses immediate triggering with a modified version of the runtime system in which threads access their deques in FIFO order with

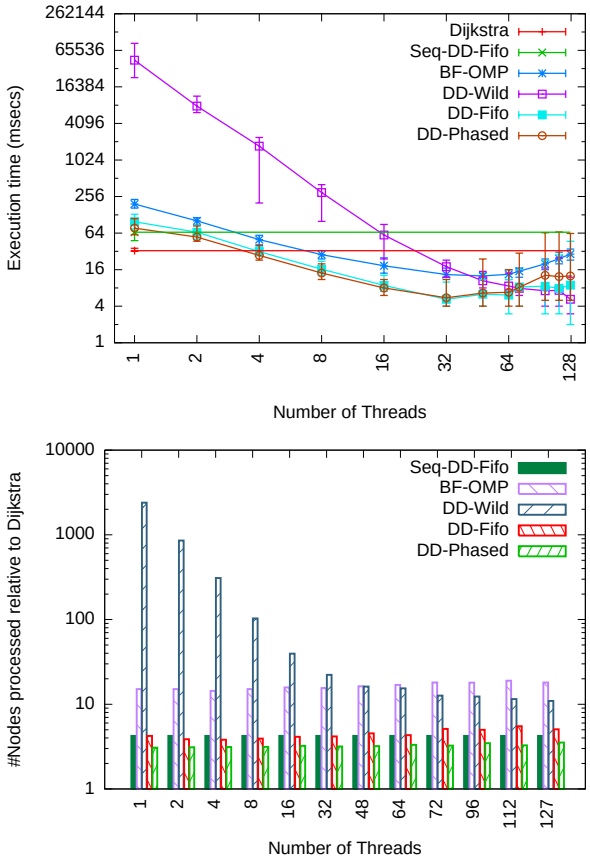

Figure 3: Top: SSSP execution time (mean of 10 runs with different source nodes, error bars showing min/-max). Bottom: Work performed, normalized to Dijkstra.

self-stealing (pushing on one end, but with workers and thieves popping from the other end).

Figure 3 compares these data-driven algorithms with three alternative algorithms: BF-OMP is an OpenMP-based parallel version of the Bellman-Ford algorithm that avoids write-write conflicts by having a task relax all *incoming* edges of a node. Dijkstra is the classic sequential SSSP algorithm. Seq-DD-Fifo is a sequential version of DD-Fifo that uses a sequential queue to store the nodes to be processed instead of our abstractions (we used it to estimate the overhead imposed by our runtime system). The figure uses the ca-HepPh collaboration network from SNAP [23] (12,008 nodes and 237,042 edges). Graphs from a variety of benchmark suites show qualitatively similar results. Results are shown in a log-log scale.

As these results illustrate, the order in which tasks are executed has a huge effect: For single-thread runs, DD-Wild, which runs tasks in LIFO order (i.e., traverses the graph depth-first), is 250x slower than BF-OMP, whereas DD-Phased and DD-Fifo, which approximate a breadth-first traversal, are about 2x *faster* than BF-OMP. Dijkstra, which relaxes nodes in an optimal order

and avoids the overhead of spawning and synchronizing tasks, is about 6x faster than BF-OMP. Comparing Seq-DD-Fifo and DD-Fifo shows that the overhead of our system is about 50%.

The data-driven algorithms all scale well up to 32 threads, with DD-Phased and DD-Fifo slightly improving their lead over Bellman-Ford (and outperforming Dijkstra with 4 or more threads), and DD-Wild rapidly catching up. Although DD-Fifo is 27% slower than DD-Phased with one thread, DD-Fifo outperforms it by 42% at 127 threads. Perhaps surprisingly, DD-Wild exhibits superlinear speedup, and with 72 or more threads, it outperforms all the other algorithms.

We can understand these results better by looking at Figure 3 (bottom), which shows the total amount of *work* done by each algorithm, measured by the number of nodes processed. With a single thread, DD-Wild does 500x more work than DD-Fifo, and 125x more than BF-OMP, its closest competitor, because of its LIFO execution of tasks. With more threads, the LIFO execution is increasingly disrupted by work-stealing, resulting in a better order for computing SSSP, and a corresponding reduction in total work. For all other parallel algorithms, the work increases slightly as the thread count increases, with DD-Phased doing the least work at all thread counts, explaining its relatively good performance at low thread counts. However, at high thread counts, the waiting due to phases limits its scalability, and DD-Fifo and DD-Wild outperform it. Although DD-Wild does more work than DD-Fifo, it outperforms DD-Fifo at high thread counts because it avoids the synchronization overheads that DD-Fifo incurs due to self-stealing.

4.2 Community-based Graph Clustering

Our final example is a clustering algorithm that partitions a graph into *communities*, each comprising nodes that are relatively strongly connected to each other and relatively weakly connected to nodes in other communities. We build on the sequential round-based algorithm of Raghavan et al. [20] (Serial). Each round, the algorithm iterates over all nodes in a random order, and assigns each node to the community with a plurality of its neighbors (ties are broken arbitrarily). The algorithm terminates after a round in which no node changes community.

OMP is a parallel version of this algorithm that uses an OpenMP parallel for loop to iterate over the nodes at each round. Both Serial and OMP can perform a lot of unnecessary work: a node changes its community in a round only if at least one of its neighbors changed its community recently (in the last or the current round), but these algorithms recompute each node's community every round even if none of its neighbors has since changed its community. Tseng and Tullsen made simi-

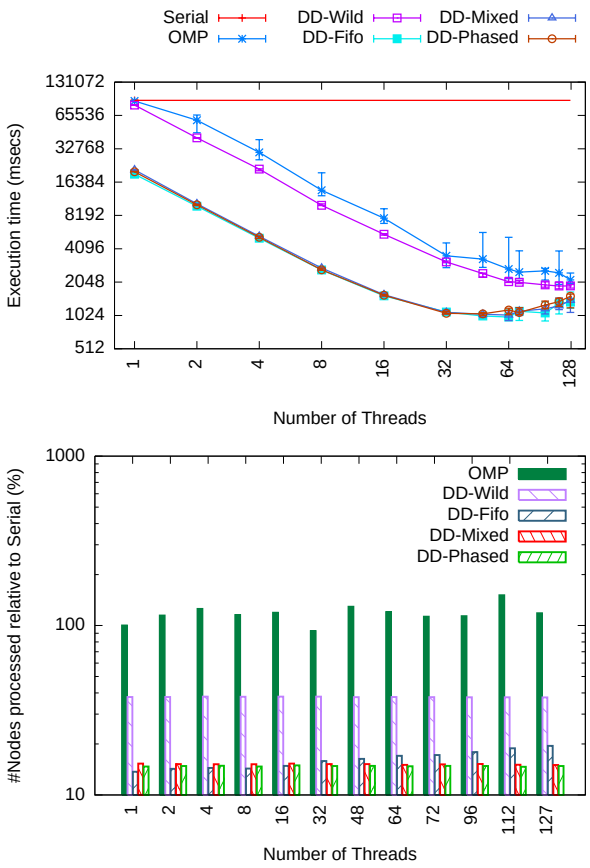

Figure 4: Top: Communities clustering execution times (mean of 5 runs, error bars showing min/max). Bottom: Work performed, normalized to Serial.

lar observations for other applications in their work on data-triggered threads [25, 26].

Based on this observation, we built four data-driven variants of this algorithm: DD-Wild, DD-Fifo, DD-Phased, and DD-Mixed. In all of them, updating a node's community triggers tasks for its neighbors. To begin the computation, we use a parallel for loop to assign a unique community to each node, triggering computation across the graph. The first three variants are similar to the corresponding SSSP variants. In DD-Mixed, a task to process a node is triggered in the current phase if that node has not already been processed in the current phase; otherwise, the task is deferred to the next phase. This more closely imitates Serial, with DD-Mixed's phases corresponding to rounds in Serial.

We ran all the algorithms on various SNAP [23] graphs, using the *modularity* metric [16] to confirm that the "quality" of the results achieved by the parallel algorithms are similar to or better than those of Serial. The top chart of Figure 4 shows, on a log-log plot, the execution times for the amazon0505 graph (10,236 nodes, 3,356,824 edges, converted to an undirected graph by

adding the reverse edges, as the above algorithm requires an undirected graph). The data-driven algorithms outperform Serial and OMP algorithms in every case. As with SSSP, constraining task execution order substantially improves performance. In particular, DD-Wild performs considerably worse than all other data-driven variants. This is explained by the bottom chart of Figure 4, that shows that, at all thread counts, DD-Wild performed significantly more work than the other data-driven variants (about 40% versus 15%, normalized to Serial). Single-threaded OMP does the same amount of work as Serial, and at higher thread counts, it usually even does 14–50% more work than Serial. This is why it performs worse than DD-Wild.

Because DD-Mixed was designed to imitate the behavior of Serial, we also compared the number of rounds for Serial to converge and the number of phases taken by DD-Mixed. Indeed, the two numbers were very similar for most graphs. However, for one graph, web-BerkStan, the number of serial rounds was an order of magnitude higher than the number of data-driven phases, regardless of the random order in which Serial processed the nodes, resulting in a serial execution time that is 40x longer than that of DD-Mixed. This suggests that for some graphs, the data-driven order of processing can lead to faster convergence than that achieved with most random orders.

4.3 Betweenness Centrality (BC)

BC is a measure of the importance of each node in a graph in terms of the number of shortest paths going through that node. Hong et al. wrote a Green-Marl program (bc-GM) that approximates BC [10], based on an algorithm by Madduri et al. [15]. Briefly, bc-GM does a BFS traversal rooted at a randomly selected node, recording the number of shortest paths from the root to each node, and then does a reverse-BFS (rBFS) traversal, recording at each node the number of shortest paths from the root to other nodes that go through it. This process is repeated several times.

We developed a data-driven variant (bc-DD) in which the BFS traversals are done using task group phases, and the rBFS traversals are done in classic data-driven fashion. The BFS and rBFS tasks in bc-DD can be overlapped to some degree, improving overall parallelism. Note that, in contrast to SSSP, executing the tasks in phases is required *for correctness*, to ensure BFS traversal.

The Green-Marl compiler does source-to-source transformation, converting a Green-Marl program into an equivalent OpenMP program. The resulting bc-GM OpenMP program dynamically constructs sets of nodes visited at each level in the BFS traversal, and within each BFS level, employs an OpenMP parallel for loop to visit each graph node at that level. We believe that our experimental comparison between bc-GM and bc-DD reflects trade offs between the OpenMP and CDDP implementations, rather than Green-Marl and CDDP abstractions. Furthermore, because OpenMP provides numerous options for how loop iterations are scheduled, we experimented with the *static* scheduling policy, the *dynamic* scheduling policy with multiple for loop chunking factors.

We conducted experiments with several input graphs and report performance results for two graphs–a social network graph (soc-LiveJournal1) and a web graph (web-Google)–that represent behaviors of all the graphs we have experimented with. Performance results are shown in Figure 5. We show the performance of the static scheduling policy (bc-GM), and the dynamic scheduling policy (bc-GM-dy-*) with different loop iteration chunking factors (128, 1K, and 4K). The chunking factors help us understand the potential impact of varying task granularity. bc-DD does not employ any chunking. We note that, unlike other benchmarks reported here, CDDP's data driven execution model does not reduce the amount of real work done in BC.

Because bc-DD tasks are far more fine grained (one task per graph node) than bc-GM work items, bc-DD incurs significant overheads for single-thread runs (up to 50%). However, this overhead diminishes rather quickly with increasing thread count, presumably because of the better load balancing achieved by our work-stealing scheduler.

The scalability results are mixed. While bc-DD performs significantly better than bc-GM on the soc-LiveJournal1 graph (top Figure 5), the bc-GM-dy-* versions with coarse chunking appear to outperform bc-DD by a small margin. Both bc-DD and the coarse-grained bc-GM-dy-* versions outperform bc-GM, presumably because they offer better load balancing. The finer-grained bc-GM-dy-128 appears to suffer with the consequences of too fine a granularity, which leads to worse communication to computation ratios on our experimental platform.

Interestingly, for web-Google (bottom Figure 5), bc-GM appears to perform best. We believe this to be a function of the graph structure. In particular, because the average degree of graph nodes in web-Google is much smaller than soc-LiveJournal1 graph nodes, the amount of work done in each bc-DD task for webGoogle is correspondingly much smaller. This significantly increases communication to computation ratio for the dynamic scheduling schemes on webGoogle. Because OpenMP static scheduling comes with essentially no communication between worker threads (except for communication happening via application data), it appears to outperform all the dynamic scheduling alternatives (including bc-DD), all of which encounter communication overheads while enforcing dynamic scheduling. bc-DD appears to

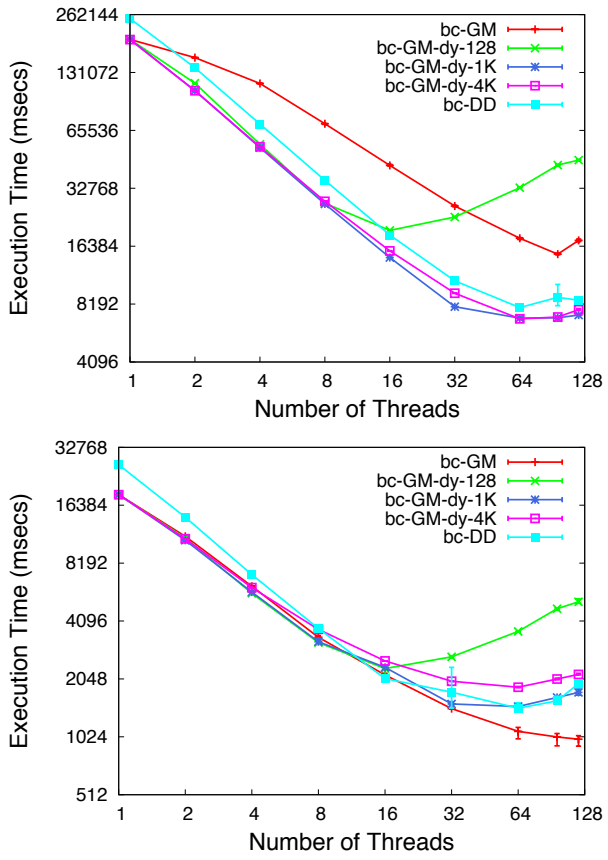

Figure 5: Betweenness Centrality performance results on two graphs: (i) a social network graph (soc-LiveJournal1), with approximately 4.8 million nodes and 137 million edges (top half); and (ii) a web graph (web-Google) with approximately 1 million nodes and 10 million edges (bottom half). Both the graphs, taken from the SNAP dataset [23], were converted to undirected graphs by adding reverse edges.

be competitive with the best performing dynamic bc-GM version.

We believe that the various bc-GM versions we experimented with benefit significantly from chunking of loop iterations. However, we do not have support for such chunking in our CDDP scheduler. Implementing a strategy of dynamically consolidating several CDDP tasks into a bigger task can potentially significantly boost performance of some of our benchmarks. We plan to investigate this direction in future work.

5. Related Work

Data-flow systems. Data-driven parallelism stems from early work on data-flow computation [2, 6], in which data dependencies between instructions drive execution. We work at a coarser grain—typically, tasks access locations and perform local computation. The StarSs programming model [19] provides a form of data-flow computation using coarse-grained tasks. Synchronization is provided via barriers which wait until all spawned tasks are complete; CDDP phases avoid the need to wait for all tasks to complete before new tasks can be spawned. Gajinov et al. [8] and Seaton et al. [22] explored combinations of data-flow programming models and atomic tasks. These systems do not provide CDDP's control over the scheduling order of parallel tasks. Our current system does not require tasks to be atomic (although we discuss future work in this direction in Section 6). The Habanero project [24] includes data-driven tasks. In that model, tasks are spawned explicitly. Futures allow synchronization on tasks as they complete. In CDDP, our focus is constraining the order of tasks as they run, rather than waiting for specific tasks to finish.

Incremental and self-adjusting computation. Recent work on incremental [5, 18] and self-adjusting computation [1, 9] has explored the ability to avoid repeating unnecessary work. Tseng and Tullsen demonstrate substantial speed-ups using data-triggered threads [25, 26]. CDDP benefits from the same ability to avoid repeated work, but provides additional control over how parallel work is scheduled.

Domain-specific scheduling. Researchers have often observed that different workloads perform best under different schedulers [13, 21]. In contrast, CDDP's abstractions allow the programmer to constrain task ordering without requiring a full scheduler to be written, or requiring a single policy to be applied to a complete process.

BSP. The idea of *phases* in task groups builds on ideas in the classic BSP programming model [27], but BSP targets different kinds of computations than CDDP.

6. Conclusion and Future Work

In this paper, we have described *constrained data-driven parallelism*. In CDDP, parallel task execution is driven by changes to data, and the programmer can impose additional constraints on the order in which tasks run.

Our preliminary exploration has focused on a simple set of abstractions for expressing parallelism and for imposing constraints on execution order. Our initial results suggest that these abstractions can provide significant performance improvements. Compared with "unconstrained" approaches, CDDP enables the programmer to avoid pathologically bad scheduling orders. Compared with non-data-driven algorithms, CDDP avoids repeating computation when data does not change.

We are encouraged by these preliminary results, and hope to extend our work in several directions. We wish to study the use of CDDP for additional algorithms and workloads. Our initial exploration has focused on graph algorithms on different kinds of input (e.g., planar graphs versus small-world graphs. We would like to study a

broader range of algorithms, and to characterize where CDDP is an appropriate solution (e.g., for which of the "dwarfs" of Asanovic et al. [3] it is an appropriate fit). This study will help us identify whether our current abstractions of task groups and phases are sufficient for a wide range of algorithms, or whether additional or alternative abstractions are needed.

We have considered whether CDDP should include a notion of *atomic* tasks. These tasks would execute atomically and in isolation from one another. In addition to CDDP's existing constraints, tasks would have to execute after the atomic task that triggered them. This builds on aspects of atomic dataflow models [8, 22], and of the automatic mutual exclusion system [11]. Introducing atomicity simplifies programming by preventing tasks from observing each others' intermediate state. In addition, it is possible that implementations may be able to use one mechanism for detecting conflicts between tasks and for detecting when a new task should be spawned.

As we gain experience with CDDP, we want to refine the syntax for expressing constraints, and to investigate the trade-offs in providing language support. Introducing atomicity may push us toward compiler support. We have not yet tried to define a formal semantics for CDDP.

Finally, our initial implementation supports execution only within a single shared memory system. With increasing interest in large data sets, we plan to extend our implementation so that it supports similar programming abstractions but is not limited to use a single shared memory system. This will enable exploration of a number of interesting issues, such as adapting (perhaps dynamically) to use more systems when there is sufficient parallelism available, without requiring programmers to rewrite their applications.

References

[1] U. A. Acar. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation*, pages 1–6, 2009.

[2] Arvind and D. E. Culler. Dataflow Architectures. *Annual review of computer science*, 1:225–253, 1986.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

[4] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–28, 2005.

[5] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 85–98, 2012.

[6] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd International Symposium on Computer Architecture*, pages 126–132, 1975.

[7] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *Proceedings of the 26th Annual ACM symposium on Principles of Distributed Computing*, pages 13–22, 2007.

[8] V. Gajinov, S. Stipic, O. S. Unsal, T. Harris, E. Ayguade, and A. Cristal. Integrating dataflow abstractions into the shared memory model. In *Proc. 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '12, pages 243–251, 2012.

[9] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 25–37, 2009.

[10] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 349–362, 2012.

[11] M. Isard and A. Birrell. Automatic mutual exclusion. In *HotOS '07: Proc. 11th Workshop on Hot Topics in Operating Systems*, May 2007.

[12] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 65–76, 2009.

[13] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proceedings of the 20th annual symposium on Parallelism in Algorithms and Architectures*, pages 217–228, 2008.

[14] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT09*, 2009.

[15] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, 2009.

[16] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[17] OpenMP. `http://www.openmp.org/`.

[18] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, pages 1–15, 2010.

[19] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications*, 23(3):284–299, Aug. 2009.

[20] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physics Review E*, 76, 2007.

[21] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *ASPLOS 2010: Proc. 15th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 311–322, 2010.

[22] C. Seaton, D. Goodman, M. Luján, and I. Watson. Applying dataflow and transactions to Lee routing. In *MULTIPROG 2012: Proc. 5th Workhop on Programmability Issues for Heterogeneous Multicores*, 2012.

[23] SNAP: Stanford Network Analysis, `http://snap.stanford.edu/`.

[24] S. Tasirlar and V. Sarkar. Data-driven tasks and their implementation. In *Proceedings of the 2011 International Conference on Parallel Processing*, pages 652–661, 2011.

[25] H.-W. Tseng and D. M. Tullsen. Data-triggered threads: Eliminating redundant computation. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 181–192, 2011.

[26] H.-W. Tseng and D. M. Tullsen. Software data-triggered threads. In *OOPSLA 2012: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 703–716, 2012.

[27] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.