Towards Whatever-Scale Abstractions for Data-Driven Parallelism

Tim Harris[†] Maurice Herlihy[†] Yossi Lev[†] Yujie Liu[⋆]
Victor Luchangco[†] Virendra J. Marathe[†] Mark Moir[†]

Oracle Labs[†] Lehigh University[⋆]

Abstract

Increasing diversity in computing systems often requires problems to be solved in quite different ways depending on the workload, data size, and resources available. This diversity is increasingly broad in terms of the organization, communication mechanisms, and performance and cost characteristics of individual machines and clusters. Researchers have thus been motivated to design abstractions that allow programmers to express solutions independently of target execution platforms, enabling programs to scale from small shared memory systems to distributed systems comprising thousands of processors. We call these abstractions "Whatever-Scale Computing".

In prior work, we have found data-driven parallelism to be a promising approach for solving many problems on shared memory machines. In this paper, we describe ongoing work towards extending our previous abstractions to support data-driven parallelism for Whatever-Scale Computing. We plan to target rack-scale distributed systems. As an intermediate step, we have implemented a runtime system that treats a NUMA shared memory system as if each NUMA domain were a node in a distributed system, using shared memory to implement communication between nodes.

1. Introduction

Modern computing systems are increasingly diverse, complex and heterogeneous, ranging from shared-memory multi-socket multi-core systems, to racks of computers connected via high-speed interconnects, to loosely coupled farms of thousands of processors. Writing software that performs well across such a variety of systems is difficult: nodes may communicate via high-speed message-passing interconnects such as InfiniBand, or via shared memory, which may or may not be cache-coherent, or some combination of these; locality characteristics vary depending on system topology; even within one system, nodes may differ from each other, and may change over time. We need models that facilitate programming for such platforms by providing abstractions that can make effective use of whatever hardware happens to be executing the program; we say these models support *whatever-scale computing*.

The quest to improve programmer productivity through such models is not new. It motivated parts of the design of popular MapReduce frameworks [9, 10, 28], and recent languages for scientific computing such as X10 [7], Chapel [6], and Fortress [16], among other systems. The recent surge in domain-specific languages (DSLs) for areas such as machine learning [25], graph analytics [19, 21, 22], or statistical computation [24] reflects a similar desire to boost productivity via high-level abstractions.

This paper describes *Domino*, a whatever-scale programming model we are developing to support data-driven computation. Domino provides an abstract model of distribution and locality, along with constructs for manipulating remote data and computation, so that a single program can be mapped onto a variety of systems using implementation techniques appropriate for each system: we use a different implementation of Domino for each system, but want to avoid manual refactoring or tuning of the workloads running above it.

Domino builds on our prior work on *constrained data-driven parallelism (CDDP)* [18], which provided *triggers* that associate *handler* functions with pieces of data and spawn *tasks* that execute a handler whenever the associated data is updated. Tasks can run in parallel, and computation typically propagates as a wavefront through a data structure as updates to one piece of data structure trigger handlers that update neighboring pieces. Our earlier paper illustrated the use of this model for graph analysis workloads and a discrete event simulator [18]. Our earlier work provided additional control over how tasks are run by grouping their execution into *phases* and enabling the programmer to identify when a task should be triggered for execution in the *active* (current) phase or *deferred* to the next phase. This degree of control was crucial for obtaining good performance in several workloads.

As described in Section 2, Domino extends this model by:

- providing special *global references* to identify data that *may* be held on a remote machine and to specify how arrays should be partitioned across machines; and
- providing `async` and `do..finish` constructs, with which the programmer can express opportunities for hiding the latency of remote operations when running in large/distributed systems.

The semantics of these constructs is designed carefully so that they impose no overhead when running on a small machine, or when accessing local data.

In Section 3, we describe three different approaches to implementing these sets of constructs: one for small cache-coherent shared-memory machines, one for NUMA machines, and one for clusters without cache-coherent shared memory. We describe our implementation of two of these; the second is an intermediate step towards the third, which we have not yet implemented. This is work in progress, and we do not yet have experimental results to present. Related work and concluding remarks appear in Sections 4 and 5.

2. Programming Model

Before presenting the extensions for Domino (Sections 2.1–2.2), we briefly review our prior work on CDDP [18]. In this model, a computation is divided into *tasks* executed by threads. One task may spawn new tasks explicitly via a parallel-for loop, or implicitly by updating data which has a trigger attached to it:

```
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. 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. The functions f and g take a single argument, a pointer to the data that was modified. We provide an example of how trigger functions can be used in practice in Section 2.3.

A trigger is *deferred* if it is declared with the optional deferred keyword; otherwise, it is *immediate*. A task spawned by a deferred trigger is not executed until all tasks spawned by immediate triggers have completed. Such declarations implicitly partition tasks into a totally ordered sequence 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. At any time, every unfinished task is in either the current phase or the next phase of the entire computation. In addition, a callback function to be invoked between phase transitions can be registered with the runtime system.

2.1 Domino

The abstractions of CDDP were designed for a shared memory model in which tasks running on any thread could access any data across a single shared heap. In Domino we wish to extend this model to support NUMA machines and distributed systems, without imposing costs when running on SMP machines.

Our overall approach is to partition a program's data across a set of *compute domains*, but to retain the basic data-driven execution model, with handler functions running at the compute domain of the data they are watching. We provide three constructs for expressing and controlling accesses between compute domains:

```
// r is a global reference to an object of type T:
gRef T r;

do {
  // Functions called via global references in the async
  // block may execute asynchronously with respect to the
  // code after the async block:
  async {

    // Invoke foo at the site of r's target object. If this
    // results in a blocking RPC, then execution resumes
    // at the statement following the async block.
    r.foo();
  }
  ...
  // Execution is blocked at finish until any asynchronous work
  // started within it is complete.
} finish;
```

A gRef is an opaque global reference that enables remote data access. A function call via a global reference, if enclosed in an async block, may result in an asynchronous RPC to a different compute domain: the caller does not block for the call to return, and resumes execution at the async block's continuation. (A call via a gRef reference that is not in an async block executes synchronously, even if an RPC is required.) The do..finish block allows a task to wait for all the *pending* RPCs issued by asyncs in the do..finish block to complete and return.

We keep the async construct for handling asynchronous execution independent from the trigger mechanisms for introducing parallelism. We assume that triggers already provide sufficient parallelism to exploit the hardware available—either using the cores, or by requesting enough data accesses to saturate the interconnect. Eschewing parallelism within tasks avoids the need for synchronization within a task, simplifying the programming model and removing a source of runtime overhead when RPC calls do not block.

2.2 Data Distribution

The Domino programming model strives to enable applications to be oblivious to the actual distribution of data on the system. The programmer needs to simply specify what data may be distributed, and optionally specify the distribution pattern (blocked, cyclic, etc.). Domino provides extensions to the gRef construct to distribute large collections of objects over the entire system. The listing below illustrates these extensions:

```
gRef[dist,<pattern>] Node[] nodeArray;
do_all_gRef(foo,nodeArray,args); // runs foo in parallel on
                                 // all elements of nodeArray
... = nodeArray[i]; // returns global reference to a Node
nodeArray[i] = ...; // rvalue is a global reference to a Node
```

The dist parameter to gRef tells Domino that nodeArray is an array of gRefs that is distributed across the system (different from just an array of gRefs). <pattern> can be any of blocked, cyclic, or random. We leave exploration of richer forms of data distribution geared toward enhancing locality for future work. The do_all_gRef function, which runs a specified function over all the elements in the specified gRef array, is useful for array-wide parallel invocations of operations (e.g., initialization, reductions).

2.3 Example

Figure 1 shows pseudo-code that uses Domino to implement PageRank [4], a popular algorithm that computes the relative importance of nodes in a web graph based on transitive references made to each node in the graph. PageRank is an iterative algorithm that recomputes the new rank for each node based on the ranks of its incoming neighbors from the previous iteration. It terminates when the cumulative changes in ranks of nodes goes below a threshold, or after a specified maximum number of iterations. The algorithm is inherently data-driven because a node's rank changes in one iteration only if the ranks of at least one of its incoming neighbors changed in the previous iteration.

Recomputing each node's rank is done in two consecutive *gather* and *scatter* steps. The gather step first accumulates the ranks of all the incoming neighbors of the target node, using asynchronous calls to getRank. Thereafter, the continuation in computeRank calculates the new rank of the target node, and pushes notification updates downstream, using the deferred triggers associated with the phaseId fields of its outgoing neighbors.

These two asynchronous steps are ordered by the barrier at the end of the first do..finish block, while different phases of the algorithm are implicitly ordered via the use of the deferred triggers. The barrier at the end of the scatter step (second do..finish block) is necessary to guarantee that the current phase is not over before all tasks for the next phase are triggered.

3. Implementation

We now discuss how to implement Domino in (*i*) cache-coherent shared memory systems (Section 3.1), (*ii*) large cache-coherent NUMA systems (Section 3.2), and (*iii*) distributed, tightly coupled rack-scale computers with high speed interconnect such as InfiniBand (Section 3.3).

Our implementations are in C++, and we use various C++ classes and macros to prototype the abstractions in the Domino programming model. As in our previous work [18], computation is driven by direct and indirect trigger objects, which are implemented using templatized Trigger<T> and TriggerP<T> classes, respectively. These classes wrap the relevant object (or a pointer to an object) of type T. Each trigger object has a *handler* function that is registered with the trigger via an argument to its constructor. Triggers provide special getter and setter functions to access

```
class Node {
  ...
  int phaseId;
  // deferred trigger
  phaseId triggers deferred computeRank();

  static int globalPhaseId;      // Shared (static) variable
  ...                            // holding the current Phase Id;
                                 // Updated once per phase by the
                                 // transition callback function.
  gRef Node[] inNeighbor;
  gRef Node[] outNeighbor;
  ...
  float rank[2];

  void computeRank() {
    gRef Node neighbor;
    int sumNeighborRanks = 0;
    do {
      for (int i = 0; i < numInNeighbors; i++) {
        neighbor = inNeighbor[i];
        // Fire off a getRank call on this neighbor
        async {
          sumNeighborRanks += neighbor.getRank();
        }
      }
    } finish; // Wait until all the async work inside is done

    // d is the damping factor (typically 0.85)
    rank[(globalPhaseId+1)%2] =
      ((1 - d)/numNodes) + (d*sumNeighborRanks);

    do {
      for (int i = 0; i < n.numOutNeighbors; i++) {
        neighbor = n.outNeighbor[i];
        async {
          // a successful CompareAndSet triggers
          // computeRank() (only once) for the next phase;
          //
          CompareAndSet(&neighbor.phaseId,
                        globalPhaseId,
                        globalPhaseId+1);
        };
      }
    } finish; // Wait until all the async work inside is done
  }
  ...
  float getRank() {
    return rank[globalPhaseId%2]/numOutNeighbors;
  }
  ...
};
...
// distributed gRef array of Nodes
gRef[dist,blocked] Node[] nodes;
...
```

Figure 1: PageRank pseudo-code using Domino abstractions.

the encapsulated data. The setter functions (e.g., Set and CompareAndSet) also drive the computation by spawning a task that executes the handler function associated with the trigger. We also provide a Spawn function to explicitly spawn new tasks.

3.1 Domino-SM — Shared Memory Implementation

In our implementation for the shared memory setting, called Domino-SM, the entire Domino application resides in a single process's address space. We use a work-stealing-based scheduler to execute tasks generated by triggers. In this setting, a gRef is simply a regular object reference, and there is no need for RPCs (all called functions are directly executed). The async and do..finish constructs have no effect dynamically. Thus, we do not incur any performance overhead from the constructs that we added to Domino for distribution.

Concretely, our shared memory runtime system uses a pool of worker threads, each with a work-stealing deque [8]. Each thread gets tasks from its work deque, and adds tasks that it spawns to its work deque. If a thread's work deque is empty, the thread attempts to steal a task from the deque of another thread, selected at random. Tasks are executed in phases, which are generated by virtue of deferred triggers. We use scalable nonzero indicators (SNZI [12]) to track whether any tasks are present in each phase. SNZI objects provide scalable increment/decrement operations that avoid frequent communication between workers, as well as a fast query operation to test whether the number of applied increment and decrement operations are equal.

3.2 Domino-NUMA — Pseudo-Distributed Implementation

The last decade has witnessed a significant pivot toward multi-core multi-chip processor architectures. A natural side effect of this pivot is that machines are becoming increasingly NUMA (Non-Uniform Memory Access) in nature, where remote memory access latencies can be substantially larger than local accesses. Although today's NUMA machines support hardware cache-coherence, existing shared-memory software often requires careful tuning and control over data placement to achieve good scalability. Possible problems include access latency and contention in the underlying interconnect. These observations have prompted researchers to consider treating large NUMA machines as distributed systems [2, 3, 26], and to require the programmer to make more explicit consideration of the machine's underlying structure.

To that end, we have built a "pseudo-distributed" implementation of the Domino programming model in which we partition the runtime system between NUMA domains. Domino-NUMA shares some similarities with our shared-memory implementation, but the constructs related to creating and handling data distribution now have meaningful manifestations. The complete runtime system still operates within a single process, but we use message-passing for cross-domain operations on application data. The only shared-memory data structures used are the SNZI objects used for synchronization between phases, and the communication channels implemented as single-producer-single-consumer queues for inter-domain messages.

As with Domino-SM, a work-stealing runtime system is used to schedule the execution of tasks. With Domino-NUMA we use a separate work stealing scheduler per compute domain; all inter-domain communication is done via RPCs, and we do not migrate tasks between domains.

Split-tasks. In our current prototype implementation, the `async` and `do...finish` constructs must be expressed manually by decomposing a function such as `computeRank`. This adds two sources of complexity which we plan to avoid in the future:

- A task must be rewritten as a *split-task* consisting of a series of *task-steps*, each of which performs a piece of the function until an RPC operation or the boundaries between async and do..finish blocks. For instance, the `computeRank` function would be split into task-steps that send `getRank` calls to each in-neighbor in turn, a task-step that updates the `rank` array, and then further task-steps to execute the scatter phase, that invokes `CompareAndSet` on the out-neighbors. This manual transformation is laborious, but it could be performed automatically by a compiler (as it was in prior work using these constructs [17]).
- Our implementation exposes more parallelism than intended in our programming model. In particular, multiple task-steps from

the same original task may run concurrently on different worker threads (for instance, in the implementation of `computeRank`, multiple increments to the same element of the `rank` array may run on different threads).

This means that, with our current implementation, the programmer must use an atomic increment rather than an ordinary increment. In the future we intend to avoid this by ensuring that the task-steps from a single task do not become split over multiple workers—this would be consistent with our intended informal semantics in Section 2, and with the original operational semantics for AC [17].

Initially, the trigger handler spawns a split-task at its first task-step. Task-steps are coordinated via a SetContinuation method which defines a new task-step to add to a worker's deque if the current task-step blocks. For instance, in `computeRank`, this method is used to let the next iteration of the two loops execute whenever an RPC in one of the loops blocks. These continuation functions, along with call-backs when RPC results are received, allow the program to track the number of outstanding asynchronous RPC calls, and to only run work after a `do..finish` once all of the results have been received. Note that a split-task will never span more than one phase in the data driven computation, as it is merely a task broken into multiple pieces, to allow worker threads to execute other split-tasks' steps while waiting for an RPC to return.

Cross-domain RPC. We implement the gRef construct and its distributed array variant in Domino-NUMA using C++ templatized classes GRef<T> and GRefArray<T> respectively. Internally, GRef<T> encapsulates the compute domain ID and a pointer within that domain's address space. (This information is recorded when the object referenced by gRef is allocated in a particular domain.) GRefArray<T> maintains a map of index ranges and IDs of domains hosting those ranges, along with the starting address of each range within each domain. We implement the cross-domain RPC operations via a Run method on each GRef<T>. This method takes a per-object-type function ID, input arguments, a pointer to a local buffer to hold the return result, and an optional function ID for a callback to run locally once the result is received.

The RPC mechanism uses two additional threads per domain, one for receiving incoming requests, and one for sending outgoing requests. A worker thread posting an RPC sends a request to the local sender thread, which forwards the request to the remote receiver service thread via a dedicated channel (implemented as a fixed-sized circular single-producer-single-consumer buffer). On receiving a request, the receiver service thread instantiates a new split-task for the request and posts it in a single-producer-multi-consumer queue, to be processed by the worker threads.

Once the split-task is executed, the completion notification, along with the response buffer, are sent back to the original requesting domain. A receiver at the requester's domain picks up the response and directs it to the original requester split-task, where the pending RPC counter is decremented atomically (once this counter goes down to 0, the split-tasks next step will be executed), or a new split-task is created to execute the request's callback function if there is one (the RPC counter is decremented once the callback returns).

Phase tracking. Finally, we adapted the phase tracking mechanism to Domino-NUMA by adding a cross-domain level to the hierarchy used in the SNZI objects. In particular, each domain maintains SNZI object(s) to indicate whether there are pending tasks *in that domain* for the current and next phases. In addition, a system-wide counter for the number of "active domains" is shared between all domains, and is incremented or decremented by a domain when its SNZI object changes its value from 0 to 1 or 1 to 0, respectively. In Domino-NUMA, the active domains counter can be shared between domains without the risk of significant inter NUMA-node cache coherence traffic because the counter is updated infrequently. In a real distributed setting a specialized software coherence mechanism will likely be sufficient for this counter.

3.3 Domino-IB— Tightly Coupled Clusters

Driven by the explosion in the amount of data modern software needs to process, tightly-coupled distributed systems have gained significant traction in the computing industry [13, 14]. These systems typically comprise clusters of machines with high speed interconnects, such as InfiniBand, where inter-node communication is via RDMA or explicit message passing interfaces.

We plan to create a further Domino implementation to target such clusters. Because Domino's abstractions have been explicitly designed to separate application code from the organization and communication details of the underlying target systems, we believe that the implementation changes will be relatively modest.

Compared with Domino-NUMA, we would need to replace the RPC mechanism with one that can operate over the cluster's interconnect, and would need to design a distributed form of the SNZI objects used to control the switch-over between phases.

4. Related Work

Domino builds on many areas of related work, which we briefly discuss here.

The Tera architecture [1] and Cray XMT [15] support large numbers of hardware threads and overlap execution of one thread with the servicing of data accesses from other threads. This approach is effective for workloads with poor locality but large degrees of parallelism in which an alternative cache-based architecture would be ineffective. This observation has recently been explored in software in systems such as Grappa [22] in which requests to access remote data are aggregated into larger bulk requests that can be transferred efficiently over InfiniBand. Compared with Grappa, the Domino programming model aims to support efficient non-distributed implementations via the design of the semantics of gRef references and the async and do..finish constructs to allow these to be elided.

As with Grappa, Domino has been designed to be effective for workloads with poor locality by making large numbers of tasks available for execution. In Domino-NUMA, splitting work into task-steps enables a worker thread to switch between pieces of different tasks while waiting for an RPC to complete.

Partitioned global address space (PGAS) languages provide programming models in which a program's shared data is partitioned between a set of processors or threads [6, 7, 11, 16, 23]. Each can access its own local portion of the data, or make remote accesses to data held elsewhere. Typically, the programming language provides abstractions to identify which data is local or remote, or to request code to execute at the "home" of particular data. Domino takes a similar approach, providing a PGAS language built around data-driven parallelism.

The Barrelfish [2] and fos [27] research operating systems have explored the use of distributed systems techniques within a single machine. Several researchers have examined the trade-offs in multi-processor machines between using shared memory directly, or using it to provide an efficient implementation of message passing [2, 5, 20]. This "pseudo-distributed" approach motivated our exploration of Domino-NUMA.

The async and do..finish constructs are based on those from the Asynchronous C language extensions used in Barrelfish [17]. As with AC, we chose to keep these constructs separate from those used to introduce parallelism (although, as we described in Section 3, our current prototype implementation currently requires

manual synchronization to achieve this). Our decision is motivated by the desire to allow the constructs to be elided completely in implementations such as Domino-SM. In contrast, X10 uses an `async` construct to spawn work to execute in a concurrent thread.

5. Current Status

This paper introduced the Domino programming model, and described our approaches to implement it on three kinds of systems. As we reported last year [18], our shared memory implementation is complete, along with an initial set of workloads. That work illustrated the power of our abstractions to significantly curtail wasted work and boost performance of several important workloads. Domino enables similar benefits in a distributed setting, which is a significant differentiator from related work on distributed programming models [6, 7, 9–11, 16, 22, 23, 28]. Our second implementation, treating a NUMA machine as a "pseudo-distributed" system, is currently in progress. We have built the basic RPC mechanism, but are currently working on optimizations (such as aggregating remote requests, as is done in systems such as Grappa [22]), automating the splitting of tasks into individual steps, and avoiding unintended parallelism between steps from within the same task. We hope to use this as the basis for our third implementation, for clusters, replacing the RPC mechanism and designing a distributed SNZI protocol.

Acknowledgments

We thank Alex Kogan, Daniel Goodman, and our anonymous reviewers for their feedback on earlier drafts of the paper.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, pages 1–6, 1990.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 29–44, 2009.

[3] A. Baumann, S. Peter, A. Schupbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your computer is already a distributed system. Why isn't your OS? In *5th USENIX Workshop on Hot Topics in Operating Systems*, 2009.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 107–117, 1998.

[5] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. J. Marathe, and M. Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *OPODIS '13: Proc. Principles of Distributed Systems - 17th International Conference*, pages 83–97, 2013.

[6] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

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

[8] 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.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, 2004.

[10] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818, 2010.

[11] T. El-Ghazawi and L. Smith. Upc: Unified parallel C. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, 2006.

[12] 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.

[13] Oracle Exadata Database Machine, `http://www.oracle.com/us/products/database/exadata/overview/index.html`.

[14] Oracle Exalogic Elastic Cloud, `http://www.oracle.com/us/products/middleware/exalogic/overview/index.html`.

[15] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *Proceedings of the 2Nd Conference on Computing Frontiers*, CF '05, pages 28–34. ACM, 2005.

[16] The Fortress Programming Language, `https://projectfortress.java.net/`.

[17] T. Harris, M. Abadi, R. Isaacs, and R. McIlroy. AC: Composable Asynchronous IO for Native Languages. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 903–920, 2011.

[18] T. Harris, Y. Lev, V. Luchangco, V. J. Marathe, and M. Moir. Constrained Data-Driven Parallelism. In *5th USENIX Workshop on Hot Topics in Parallelism*, 2013.

[19] 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.

[20] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 6–6, 2012.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[22] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching Large Graphs with Commodity Processors. In *3rd USENIX Workshop on Hot Topics in Parallelism*, 2011.

[23] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *ACM FORTRAN FORUM*, 17(2):1–31, 1998.

[24] The R Project for Statistical Computing, `http://www.r-project.org/`.

[25] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proceedings of the 28th International Conference on Machine Learning*, pages 609–616, 2011.

[26] D. Wentzlaff and A. Agarwal. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[27] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *SOCC '10: Proc. 2010 Symposium on Cloud Computing*, pages 3–14, June 2010.

[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 2–2, 2012.