Programming Large Dynamic Data Structures on a DSM Cluster of Multicores*

Sai Charan Koduru, Min Feng and Rajiv Gupta

University of California, Riverside
email: {scharan, mfeng, gupta}@cs.ucr.edu

Abstract

Applications in increasingly important domains such as data mining and graph analysis operate on very large, dynamically constructed graphs, i.e. they are composed of dynamically allocated objects linked together via pointers. Parallel algorithms on large graphs can greatly benefit from software Distributed Shared Memory's (DSM) convenience of shared-memory programming and computational scalability of multiple machines. However, prior DSM systems did not provide programming and memory abstractions suitable for working with large dynamic data structures.

In this paper, we present *dynamic DSM* (dyDSM), an object-based DSM that targets applications which operate on large dynamic graphs. To achieve this goal, dyDSM differs from traditional DSM systems in four important ways. First, dyDSM provides programming abstractions that expose the structure of the distributed dynamic data structures to the runtime enabling the communication layer to efficiently perform *object level data transfers* including object prefetching. Second, the dyDSM runtime enables the management of large dynamic data structures by providing various memory allocators to support parallel initialization and storing of dynamic data structures like graphs. Third, dyDSM provides support for exploiting speculative parallelism that is abundant in applications with irregular data accesses. Fourth, dyDSM runtime actively exploits the multiple cores on modern machines to tolerate DSM access latencies for prefetching and updates. Our evaluation on a 40-core system with data mining and graph algorithms shows that dyDSM is easy to program in the very accessible C++ language and that its performance scales with data sizes and degree of parallelism.

1 Introduction

Clusters offer horizontal scalability both in terms of number of processing cores and the amount of memory. They are especially attractive for modern applications because, with the advances in network technology, a cluster can provide an application with a faster storage system than a disk based storage system [7]. While distributed memory clusters are more powerful than the shared memory machines they are composed of, programming them is challenging. While PGAS languages [14, 6, 12] ease distributed programming, they primarily explore parallelism for array-based data-parallel programs using data partitioning. For the most part they do not consider dynamic data structures nor do they explicitly support software speculation.

We have developed dyDSM which targets applications that employ large dynamic data structures and overcomes the inadequacies of prior DSM systems. Specifically, we target applications that operate on large dynamic data like graphs that exhibit no spatial locality and can exploit speculation for speedup. dyDSM addresses these issues by providing:

*This research was supported in part by a Google Research Award and NSF grants CCF-1318103, CCF-0963996 and CCF-0905509 to the University of California, Riverside.

1. Programming abstractions to communicate structure of the distributed data structure to the compiler;

2. Multiple allocators to perform parallel and local initialization and distribution of data structures;

3. Programming abstractions to effectively express speculation in programs and a runtime to transparently perform speculation on clusters; and

4. A runtime to actively exploit the multiple cores present on individual machines in the cluster.

Like PGAS languages, dyDSM automatically manages the distributed address space, but adds language and runtime support for speculative execution. We now describe the application features of interest and indicate the areas where prior solutions fall short.

Absence of Spatial Locality. Previous DSM systems (virtual memory and cache coherence protocol inspired systems [21] and Single-Program, Multiple Data [5] systems) were not designed to support dynamic data structures, did not support speculative parallelism, and did not consider multicore machines. They mostly targeted applications that use large arrays that exhibit spatial locality while the applications we study extensively use dynamically allocated data structures that lack spatial locality. Therefore caching and software cache coherence protocols [21] that they employ are ineffective in tolerating communication latency for the irregular applications we target.

dyDSM effectively deals with lack of spatial data locality in irregular applications: programming annotations expose the structure of the distributed dynamic data structures to the runtime enabling the communication layer to perform *object level data transfers* including object prefetching. By avoiding the use of traditional caching protocols, the need for cache coherence protocols and thus their drawbacks are eliminated.

Large Dynamic Data Structures. Previous object-based systems like Orca [1] required the programmer to explicitly and statically partition and allocate data across cluster. The computations were then executed on the site that contained the data. Given the absence of spatial locality in graph-like dynamic data structures, there is no intelligent way for the programmer to statically partition the data. Therefore, dyDSM takes this burden off the programmer by providing allocators that automatically and uniformly distribute the data across the cluster. Further, with *large* dynamic data structures, initializing the data in parallel is also very important. Otherwise, the data initialization could itself become a bottleneck. For this, dyDSM provides specialized allocators that enable parallel initialization and I/O of large data. Finally, dyDSM also provides an allocator to allocate data locally, in cases where the programmer knows in advance that some data is mostly used locally, and only occasionally needed on other machines in the cluster. The object transfers between thread-private memory and virtual shared memory are managed automatically by dyDSM.

Speculative Parallelism for Irregular Applications. Exploiting parallelism in irregular applications often also requires the use of speculation [26]. With irregular programs being our targeted applications, dyDSM eliminates the need for general purpose caching and coherence by employing *thread-private memory for the copy-in copy-out speculative model* [13]. The DSM is organized as a *two-level* memory hierarchy designed to implement the *copy-in copy-out* model of computation [13] that supports speculation but does not require use of traditional memory coherence protocols. As shown in Figure 1, the second level of DSM, named *virtual shared*

memory, aggregates the physical memories from different machines to provide a shared memory abstraction to the programmer. The first level of DSM consists of *virtual thread-private memories (PM)* created for each of the threads.

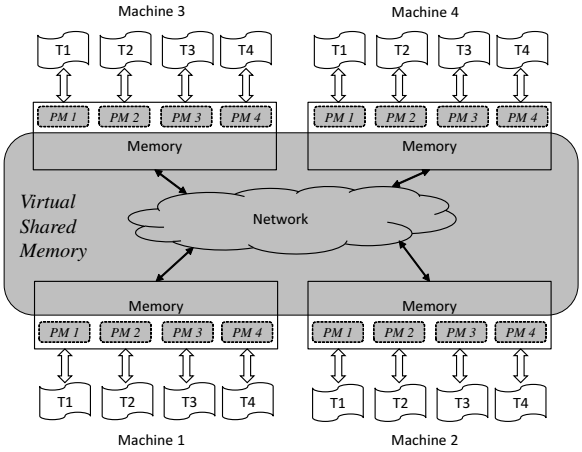

Figure 1: Memory hierarchy of dyDSM on a modern cluster.

To perform a computation, each thread must first *copy-in* the required data from the virtual shared memory into its virtual private memory. Then the thread can continue to use the local data copies in its private memory to perform computations. The *copy-out* of modified data from the private memory to the virtual shared memory makes it visible to all other machines. This computation model has two advantages. First, when a thread performs its computation using its private memory, it is able to exploit the temporal data locality *without generating network traffic*. Second, this model makes it easy to write programs that exploit *speculative parallelism* found in applications that employ dynamic data structures. Finally, the *copy-in copy-out* operations are introduced and performed by the dyDSM compiler and runtime, i.e. they do not burden the programmer. The programmer simply indicates the need for speculation by marking regions of code as *atomic* and dyDSM automatically handles all the above details.

Exploit Multicores for Tolerating Communication Latency. None of the previous DSM systems (eg. ORCA [1], Shasta [22] or Emerald [15]) were designed to utilize the modern multicore machines. Users can overcome this limitation by treating each processor core as a node in the DSM system and run multiple parallel computation tasks on each machine. However, due to the limited network resources on each machine, the performance may not scale well with the number of cores used from each machine. Since multicore machines are widely used today, dyDSM is designed to efficiently use of all of the available cores. Specifically, dyDSM utilizes a subset of available cores for asynchronous, off-the-critical-path communication, thus better hiding network latency than possible with previous approaches.

We have developed a prototype of the dyDSM system. The user writes programs in C/C++ augmented with dyDSM programming abstractions. The code is translated via dyDSM compiler based upon LLVM [19] to generate parallel code that calls the dyDSM runtime to effect allocation of objects in DSM and perform data transfers between thread-private memory and virtual shared memory. The runtime also implements speculation. We evaluated dyDSM on a cluster consisting of five eight-core machines using seven applications, including data mining applications (K-means Clustering, PageRank, Betweenness Centrality) and widely used graph algorithms (Delaunay Refinement, Graph Coloring, BlackScholes, Single Source Shortest Path).

These applications process large volumes of data. Our evaluation using 5 machine cluster with total of 40 cores yielded average speedup of 7x.

The rest of this paper is organized as follows. Section 2 describes our programming interface. Section 3 describes representation of dynamic data structures in DSM. Section 4 describes how dyDSM implements communications and supports speculation. Section 5 provides results of evaluation. Sections 6 and 7 discuss related work and conclusion.

2 Programming Interface of dyDSM

The programming interface of dyDSM does not place any significant burden on the programmer as it is like an interface for a shared memory system with some easy-to-use augmentations. dyDSM provides a familiar, shared memory-like API that allows creation and manipulation of dynamic data structures with ease. The augmentations perform the important task of identifying the presence and communicating the structure of a distributed dynamic data structure to the dyDSM runtime. They also indicate the form of parallelism, speculative or non-speculative. We now describe these programming extensions for use in C/C++ programs.

I. Structure declaration. Structure declarations are used to expose the details of the distributed dynamic data structures to the runtime enabling the communication layer to perform object level data transfers including object prefetching. In dyDSM, dynamic data structures are declared just like in shared-memory programs. The programmer need only add the `__dsm` annotation to the type declaration of an object type and the pointer variables that point to that object type. As an example, Figure 2 shows how a graph data structure must use annotation so that the created graph will reside in DSM. The `__dsm` annotation is used in two places: the definition of `struct Node`; and the array of pointers `neighbors[ ]` which links one node to its neighboring nodes in the distributed data structure. The same can be achieved if `Node` were defined as a `class`. While the annotation is simple to use for the programmer, the complex task of distributing the dynamic data structure is automatically carried out by the dyDSM runtime.

```
__dsm struct Node {
  int value;
  __dsm struct Node *neighbors[MAX_DEGREE];
};
```

Figure 2: Annotations for a distributed graph.

II. Dynamic allocation and deallocation. dyDSM eliminates the programmer burden of static partitioning and distribution of data across the cluster by providing allocators that automatically and uniformly distribute the data across the cluster. Further, with *large* dynamic data structures, dyDSM provides specialized allocators that enable parallel initialization and I/O of large data. Finally, dyDSM also provides an allocator to allocate data locally, in cases where the programmer knows in advance that some data is mostly used locally, and only occasionally needed on other machines in the cluster.

As in shared memory systems, in dyDSM, all annotated data structures are dynamically allocated and deallocated. Data distribution is achieved automatically at the allocation site by the dyDSM runtime. When an object is allocated at runtime, it is assigned to one of the machines in the cluster.

Table 1 shows the programming constructs provided by dyDSM for dynamically allocating and deallocating objects in DSM. They are similar to the `new` and `delete` operators in C++

Construct	Description	ID Format
`dsm_new Node;`	allocate an element of type `Node` and assign it to one of the machines	`[Node Type]-[Alloc. Type]-[Machine ID]-[Serial]`
`dsm_lnew Node;`	allocate an element of type `Node` and assign it to the local machine	`[Node Type]-[Alloc. Type]-[Machine ID]-[Serial]`
`dsm_cnew Node, id;`	allocate an element of type `Node` with `id` as its ID and assign it to one of the machines	`[Node Type]-[Alloc. Type]-[Customized ID]`
`dsm_delete pnode;`	deallocate the element referred to by pointer `pnode`	`--`

Table 1: Programming constructs for element allocation and deallocation.

except they allocate space in the DSM.The `dsm_new` construct randomly assigns the allocated element to one of the machines for load balancing. The `dsm_lnew` construct assigns the allocated element to the local machine. It is used to avoid unnecessary communication when the element is mostly accessed by the local machine. This is true if data is well partitioned among the machines. Construct `dsm_cnew` is designed to allow users to specify a customized id for an object. The last construct, `dsm_delete`, is used to deallocate the object referred to by the given pointer.

Next we illustrate the dynamic creation of the graph data structure declared in Figure 2 using `dsm_new`. We will illustrate the use of `dsm_lnew` and `dsm_cnew` in the next section. It is assumed that the graph is being read from a file and created and stored into the DSM. In Figure 3 the first loop (lines 4–8) reads the node information and allocates the nodes in dyDSM. The second loop (lines 9–12) reads the edge information and connects the nodes accordingly. Each invocation of `dsm_new` (line 6) assigns a given node to a random machine in the cluster. Array `node_ptrs` is a local variable that stores all node IDs, which are later used for adding edges.

```
01 // single thread builds the graph
02 if ( threadid == 0 ) {
03   begin_dsm_task(NON_ATOMIC);
04   for(i=0; i<nodes; i++) {    // read nodes
05     read_node(file, &id, &v);
06     node_ptrs[id] = dsm_new Node;
07     node_ptrs[id]→value = v;
08   }
09   for(i=0; i<edges; i++) {    // read edges
10     read_edge(file, &id1, &id2);
11     node_ptrs[id1]→nbrs.add(node_ptrs[id2]);
12   }
13   end_dsm_task();
14 }
```

Figure 3: Reading graph from a file and storing it in DSM.

III. Expressing Parallelism. Parallelism is exploited by creating multiple threads which execute the same code; however, they operate on different parts of the data structure. While this is similar to prior approaches for expressing parallelism, the difference in the code arises due to use of *copy-in copy-out* computation model [13]. Each thread executes blocks of code marked using `begin_dsm_task()` and `end_dsm_task()` in *copy-in copy-out* fashion. During the course of executing a code block the thread-private memory is used and any data not found

in the memory is copied into it. Thus the execution of a code block by a thread is carried out in *isolation* from other threads. At the end of the code block's execution the results are committed to the owners in virtual shared memory and made visible to all other threads. Note that programmer does not explicitly specify the data transfers but rather they are achieved via compiler introduced calls to the dyDSM runtime. Finally, the programmer can specify whether the update of shared memory must be `ATOMIC` or can be `NON_ATOMIC`. In the former case, atomic update guarantees sequential consistency at the code block level. It is used to implement speculative parallelism such that in case of misspeculation, commit fails and the execution of code block is reexecuted. The dyDSM runtime that implements virtual shared memory is responsible for checking atomicity.

```
01 // each thread works on a subset of nodes
02 for(i=tid*stride; i<(tid+1)*stride; i++) {
03   begin_dsm_task(ATOMIC);
04   nodeset = NULL;//construct subset of nodes
05   nodeset.insert(node_ptrs[i]);
06   while( (node = nodeset.next()) != NULL)
07     for(j=0; j<node→neighbors.size(); j++)
08       if ( check_cond(node→neighbors[j]) )
09         nodeset.insert(node→neighbors[j]);
10   update(nodeset); // update info.  in nodes
11   end_dsm_task();
12 }
```

Figure 4: Parallel computation using speculation.

Let us next illustrate the use of code blocks by threads and the use of `ATOMIC` and `NON_ATOMIC`. If we examine the example in Figure 3, because only a single thread constructs the graph, `NON_ATOMIC` is used. Figure 4 illustrates the use of `ATOMIC`. It shows a loop that performs computation on a pointer-based data structure. Similar loops can be found in many applications such as Delaunay refinement, agglomerative clustering [18], and decision tree learner [24]. In this loop, each thread executes a set of iterations of the outer for loop based upon its `tid`. A single iteration forms a code block that is executed atomically. During an iteration a node set is constructed from a given node in the data structure and then the computation performed updates the information in the node set. Although each iteration mostly works on different parts of the data structure, possible overlap between set of nodes updated by different threads (i.e., speculative parallelism) requires the use of `ATOMIC`. Note that it is difficult to write the above parallel loop using distributed memory programming model such as Message Passing Interface (MPI) since it requires developers to manually synchronize the pointer-based data structure across different memory spaces. On the other hand, dyDSM makes it easy to write the above parallel loop since the pointer-based data structure can be shared through reading and writing in the DSM.

3 Distributed Data Structures in dyDSM

In dyDSM, dynamic data structures are distributed in the virtual shared memory across multiple machines and objects on one machine can be logically linked to those on other machines. Moreover, as threads access and operate on objects of the data structure, these objects are copied into the thread-private memory. The same object may be present in the private memory of multiple threads. Figure 5 shows a graph data structure in virtual shared memory and copies of subsets of nodes in thread private memory on two machines. Let us discuss how a distributed data structure is represented and how the operations on it are implemented. The implementation of the dyDSM communication layer will be discussed in section 4.

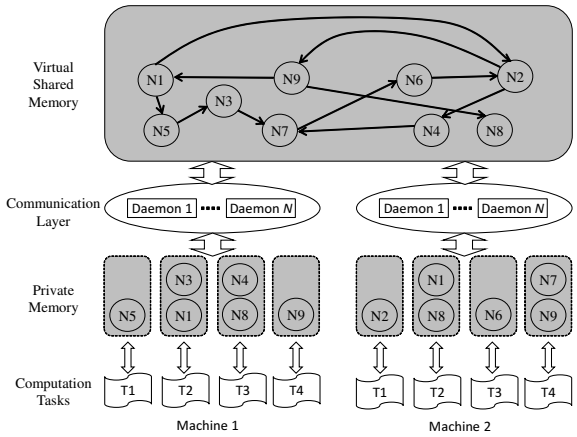

Figure 5: An example of dynamic data structure stored in dyDSM.

The compiler translates the annotated data structures and pointers by introducing calls to dyDSM runtime at appropriate places so that data needed by a thread is made available in its thread-private memory via data transfers performed by dyDSM runtime. The compiler instruments each annotated data structure to ensure when an object is accessed, the runtime is called to check if the object is in the thread-private memory. If not, data transfer is performed to copy the data object from the remote memory to the thread-private memory. All subsequent accesses to the same object will be directed to the copy in the private memory.

The dynamically allocated objects can only be accessed via pointers. Each object is assigned a unique ID that acts as its virtual address and is used to access it. Thus, the pointers are merely represented as IDs of objects they point to. When an object is allocated, its ID is generated by the runtime on the machine where the allocator is called. In dyDSM, an ID of an object is a string consisting of three parts: *object type*; *allocation type* used to allocate the object; and a *unique string* generated according to the allocation policy. The runtime fills the first two parts at the time of allocation. The unique string is determined according to the allocation mechanism used.

The construction of the unique string according to the allocation mechanism used is shown in Table 1 (third column). When `dsm_new` and `dsm_lnew` are used, the unique string is a combination of the machine ID where the allocator is called and a unique number which is generated using a local counter. When construct `dsm_cnew` is used, the unique string in the ID is specified by the programmer. This construct is designed for expressing dynamic arrays. Accessing the element using the customized ID is similar to accessing an array element in C/C++, where the structure type and customized ID are used as the array name and index.

When an ID is dereferenced, information in it is used to locate the object and then the object is copied into the thread-private memory. When committing data to shared memory, the runtime uses the object IDs to locate the owner machine. For different types of IDs (`n`, `l`, or `c`), different schemes are used to locate the machine where the object is stored. For object allocated using `dsm_new` (i.e., `n` type ID), the machine is located by hashing the ID. For object allocated using `dsm_lnew` (i.e., `l` type ID), the machine is located using the machine ID stored in the object ID. For an object allocated using `dsm_cnew` (i.e., `c` type ID), the machine is located by hashing the object ID.

In Figure 3, we demonstrated how a single thread can read a graph from a file build it using `dsm_new` and store it in DSM. Now, we demonstrate how the same task can be performed in

parallel using `dsm_cnew`. We assume that the input file is replicated on all machines. Each thread reads the entire file; however, it only builds part of the graph. In Figure 6, in the first loop (lines 3–7) each thread reads the node information and assigns each node a customized ID using `dsm_cnew`. In the second loop (lines 12–19), each thread updates the edges in the graph using the customized node IDs. With the use of customized IDs, we avoid using the array `node_ptrs` used in Figure 3 that may not fit in the local memory if the graph is extremely large. As we can see from line 12, the update of edges requires use of `ATOMIC` as multiple threads may be adding edges to the same node.

```
01 begin_dsm_task(NON_ATOMIC);
02 // read nodes
03 for(i=0; i<(threadid+1)*stride1; i++) {
04   read_node(file, &id, &v);
05   if ( i >= threadid*stride1 ) {
06     dsm_cnew Node, id;
07     Node[id].value = v;
08   }
09 }
10 end_dsm_task();
11 // read edges
12 for(i=0; i<(threadid+1)*stride2; i++) {
13   read_edge(file, &id1, &id2);
14   if ( i >= threadid*stride2 ) {
15     begin_dsm_task(ATOMIC);
16     Node[id1].nbrs.add(&Node[id2]);
17     end_dsm_task();
18   }
19 }
```

Figure 6: Reading and storing a graph in DSM in parallel via speculation.

While the code shown in Figure 6 requires use of `ATOMIC`, next we show how to perform parallel graph construction without use of `ATOMIC` using `dsm_lnew`. As before, in this version we assume that the input file is replicated on all machines. Each thread reads the file and constructs part of the graph identified to belong to its `partition`. In Figure 7 a dynamic array is created using `dsm_cnew` for holding pointers to nodes. It is used by all threads to locate nodes. However, `dsm_lnew` is used to allocate nodes. All nodes are allocated locally on the machine where they belong. When we update nodes by adding edges, each thread only updates nodes in its own partition. Thus, not only do we avoid the use of speculation, we also achieve greater efficiency due to locality.

4 Data Communication in dyDSM

Next we describe the communication layer of dyDSM and show how it implements *data transfers* and *performs communication in parallel with computation.* The communication layer also supports prefetching to further hide network latency and implements atomic updates of DSM to support speculation.

I. Communication exploiting multicore machines. In dyDSM, each multicore machine runs multiple computation threads. The performance sometimes does not scale well with the number of parallel computation tasks on each machine. To improve performance in this case, dyDSM uses a communication layer consisting of a multiple communication daemons that transfer data between the thread-private memory and the virtual shared memory (see Figure 5). The

```
01 // read nodes
02 begin_dsm_task(NON_ATOMIC);
03 for(i=0; i<nodes; i++) {
04   read_node(file, &id, &v);
05   if ( partition(id) == threadid ) {
06     dsm_cnew NodePtr, id;
07     NodePtr[id] = dsm_lnew Node;
08     NodePtr[id]→value = v;
09   }
10 }
11 end_dsm_task();
12 // read edges
13 begin_dsm_task(NON_ATOMIC);
14 for(i=0; i<edges; i++) {
15   read_edge(file, &id1, &id2);
16   if ( partition(id1) == threadid )
17   node_ptrs[id1]→nbrs.add(NodePtr[id2]);
18 }
19 end_dsm_task();
```

Figure 7: Reading a graph and storing it in DSM in parallel without using speculation.

daemons run on underutilized cores, helping the computation threads perform communication. The communication is moved off the critical path of a computation thread by offloading it to daemon threads. Thus, computation and communication is performed in parallel for improved performance.

The communication layer works as follows. On a thread's first reference to an object in virtual shared memory, the communication layer copies the object from the virtual shared memory to the thread's private memory. Subsequent accesses to the object, including both read and write, refer to the object copy in the private memory. Thus, the thread manipulates the object in *isolation*, free of contention with other threads. More importantly, caching the object eliminates the communication overhead of subsequent access to the same object. When a thread calls `dsm_end_task()`, the communication layer commits all modified objects in the private memory to the virtual shared memory allowing other threads to see the updates.

The communication daemons perform data communication in parallel with computation. When a thread needs an object, one of the daemons fetches the object from the shared memory for it. After the thread gets the object and continues its computation, the daemon can continue to prefetch the objects that will be probably accessed in the future. This can ensure that the data required by the thread is *mostly* available locally when needed. When a thread calls `dsm_end_task()`, one of the daemon also helps it perform commit so that the computation thread can continue to the next computation. Thus, the communication layer hides the network latency from the threads. As a result, the threads only need to operate on data in thread-private memory. Communication layer hides *read latency* by prefetching data needed by threads and hides *write latency* by performing commit in parallel with computation.

II. Adapting the number of communication daemons. The number of communication daemons directly impacts the performance of the system. Figure 8 shows the performance of two benchmarks with different number of communication daemons. The execution time is normalized to the one with one communication daemon. More communication daemons in graph coloring help move more communication off the critical path. However, increasing the number of communication daemons also leads to consuming more computing resources and eventually slows down the performance. For SSSP, one daemon is the best choice. We use a dynamic scheme to find the minimal number of communication daemons to use.

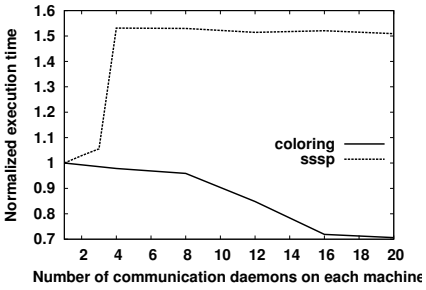

Figure 8: Performance with varying number of communication daemons.

In dyDSM, a queue between the computation threads and the communication layer holds all pending communication requests from computation threads. The queue length is used as a hint to pick the number of daemons. Large queue length means that outstanding communication requests get queued, degrading performance. Longer queue can also cause higher misspeculation rates due to delay in misspeculation detection. On the other hand, empty queue could indicate that the number of communication daemons is more than needed, wasting computing resources. In dyDSM, the program always starts with a small preset number of communication daemons. Each time a communication daemon processes a communication request, it checks the queue length. If the queue length is larger than the number of daemons, dyDSM increases the number of communication daemons by a predefined stride. If the queue length is smaller than the number of communication daemons, the number of communication daemons is decreased. To reduce the overhead of starting and terminating daemons, our implementation puts extra daemons to sleep rather than terminating them.

III. Data prefetching. When a computation task wants to access an object, it first checks whether a copy already resides in the thread-private memory. If not, the computation task waits for the remote access to fetch the data object. To avoid this wait, the communication daemons prefetch data for the computation tasks. To perform data prefetching, when a data element is accessed by a computation task for the first time, a communication daemon also predicts what elements are likely to be accessed next. If the predicted elements have not been previously copied into the local virtual memory, then the daemon sends data requests to their owners. The prediction is implemented similar to an event handler. To enable data prefetching, programmer must register a *prediction function* for the dynamic data structure. The dyDSM construct used to register the prediction function identifies the data structure requiring prefetching and the *recursion limit* (or *depth*) up to which the prediction is to be performed. When an element of the data structure is accessed, a communication daemon calls the prediction function, which takes the accessed element as input and fetches the elements that are likely to be accessed.

```
__dsm_predict(struct Node, 2)
void prediction_node(struct Node *pnode) {
   for (int i=0; i<MAX_DEGREE; i++)
     dsm_fetch(pnode→nbrs[i]);
}
```

Figure 9: An example of user-defined prediction function.

Figure 9 shows an example of a user-defined prediction function for the graph data structure used in the example from preceding section. The prediction function prefetches all 2-hop neighbors of the accessed node. This can be used in computations working on connected subgraphs,

which is true in many graph applications (e.g., Delaunay Refinement, Betweenness Centrality, Coloring).

IV. Speculation in dyDSM. Many parallel programs require speculation to perform computation correctly. For example, in graph coloring, each computation calculates the color of one node using the colors of its neighbors. It is possible for two parallel threads to read and write the color of one node at the same time; thus causing two neighboring nodes to have the same color. With speculation, all threads are assumed to access different nodes. They perform computation in isolation. When two threads actually access the same node, one thread succeeds in committing its results while the other fails and is reexecuted. With speculative execution, we also achieve sequential consistency at coarse-grained level of code regions.

Speculation requires threads to perform computation atomically and in isolation. In dyDSM, isolation comes for free since our communication layer and the use of thread-private memory already ensure computations are performed in isolation. Therefore, speculative execution does not affect the computation between `begin_dsm_task()` and `end_dsm_task()`. To ensure atomicity, speculation requires the commit to be performed atomically. When speculation is enabled, the communication layer performs commit as follows: (i) First, the daemon sends request to the virtual shared memory to lock all objects marked as read in the private memory. If locking fails for any object, it releases all acquired locks and then retries locking again. (ii) After successful locking, the daemon compares the versions of all read objects in the private memory with their versions in the virtual shared memory. If the versions do not match, it means that one of the read objects must have since been updated by another thread in the virtual shared memory. Therefore, the computation must be reexecuted. (iii) If versions match, the daemon then commits all written objects to the virtual shared memory and increases their version numbers by one. (iv) Finally, the daemon sends request to the virtual shared memory to unlock all acquired locks so that these objects can be updated by other threads. With remote communication, the commit for speculation can be time-consuming. However, dyDSM overcomes this problem by making the communication layer perform the commit in parallel with the computation; thus, hiding its overhead from computation.

5 Evaluation

This section evaluates dyDSM. The experiments were conducted on a cluster consisting of five eight-core (2×4-core AMD Opteron 2.0GHz) DELL PowerEdge T605 machines. These machines are running the *Ubuntu 10.04* operating system. They are connected through a Cisco ESW 540 switch, which is a 8-port 10/100/1000 Ethernet switch. Throughout the experiments, dyDSM always uses the memory of all five machines no matter how many machines are used for computation.

Implementation of dyDSM. Figure 10 shows an overview of our prototype implementation of dyDSM. It consists of a source-to-source compiler and a runtime library. The source-to-source compiler is implemented using the LLVM compiler infrastructure [19]. It instruments the annotated distributed data structures, translates pointers to these data structures, and generates code for prefetching. The dyDSM runtime is implemented on top of *Memcached*, an open-source distributed memory object caching system. The runtime provides support for communication, memory allocation, and speculation. In the experiments, the communication layer on each machine consists of 32 daemons. By creating many daemons dyDSM provides high network throughput. We cannot create greater number of daemons on one machine due to the size of Memcached connection pool. When no data needs to be transferred, the daemons are blocked and thus consume no processing resource.

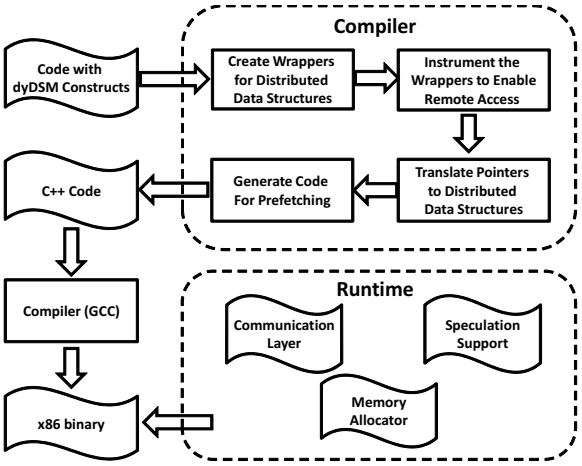

Figure 10: Overview of dyDSM prototype.

Benchmarks. To evaluate our approach, we performed experiments using seven applications, five of which require speculation. Since dyDSM is built for applications with large data sets, we selected these applications as they are data-intensive. Some of them are widely-used data mining programs designed to process large volume of data while others are graph algorithms that are used in many domains. rtabtab:bench lists the applications used, the key data structure involved and whether a speculative implementation is used. Next, we briefly describe these applications.

Benchmark	Data Struct.	Speculation?
Delaunay Refinement	Graph	Yes
Graph Coloring	Graph	Yes
Betweenness Centrality	Graph	Yes
K-means Clustering	Dynamic Array	Yes
PageRank	Graph	No
BlackScholes	Dynamic Array	No
Single Source Shortest Path (SSSP)	Graph	Yes

Table 2: Benchmark summary.

`Delaunay Refinement` implements mesh generation that transforms a planar straightline graph to a Delaunay triangulation of only quality triangles. The data set is stored in a pointer-based graph structure. `Graph Coloring` is an implementation of the scalable parallel graph coloring algorithm proposed in [4]. Each computation task colors one node using the information of its neighbors. `K-means Clustering` [10] is a parallel implementation of a popular clustering algorithm that partitions n points into k clusters. `PageRank` is a link analysis algorithm used by Google to iteratively compute weight for every node in a linked webgraph. `BlackScholes` taken from the PARSEC suite [2] uses partial differential equation to calculate the prices of European-style options. `Betweenness Centrality` is a popular graph algorithm used in many areas. It computes the shortest paths between all pairs of nodes. `SSSP` implements a parallel algorithm for computing shortest path from a source node to all nodes in a directed graph. The implementation is based on the Bellman-Ford algorithm.

5.1 dyDSM Scalability with Parallelism

Applications with high degree of parallelism can take advantage of additional cores available on multiple machines. However, the additional threads created to exploit more parallelism also stress the dyDSM communication layer. Next we study how the application speedup scales as greater number of threads are created to utilize more cores. The threads created can be scheduled on available cores in two ways: (distributed) they can be distributed across all machines; or (grouped) they can be grouped so that they use minimal number of machines. Figure 11 shows the application speedups for both the scenarios while the accompanying Table 3 summarizes the fastest execution times. The baseline is the sequential version of the application. Comparing the two curves, we observe that distributing the computation threads across the machines is a better scheduling strategy. For all benchmarks except `blackscholes`, the speedup achieved by distributing threads rises faster as the number of computation threads is increased. By distributing computation threads across the machines, idle cores left on each machines can be used by the communication layer. Therefore, the computation threads do not compete with the communication layer for cycles. For `blackscholes`, grouping threads achieves better performance because of high degree of data sharing among threads.

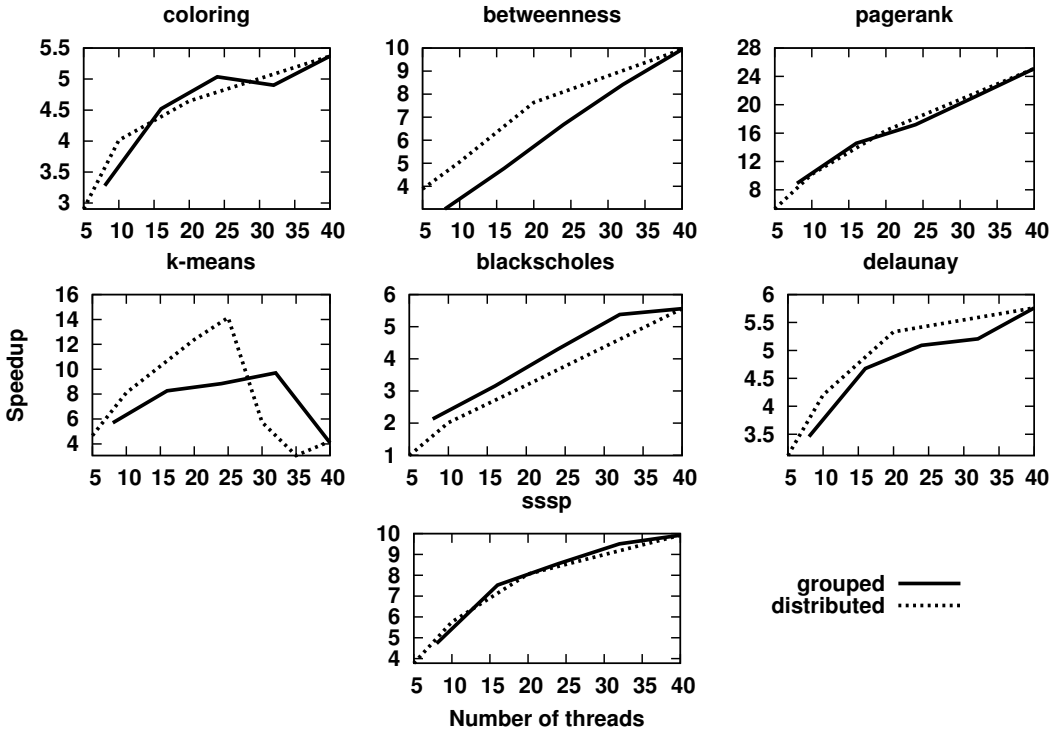

Figure 11: Speedups with varying number of computation threads.

We observe that the performance of most applications scales well with the number of computation threads. The parallel versions are always faster than the sequential version with maximum speedup of 25x. The performance of `k-means` drops with more than thirty computation threads due to very high lock contention in atomicity checks. On average, we achieve 7x speedup for the benchmarks using forty computation threads on five machines, which demon-

Benchmark	Execution time [fastest] (seconds)	Sequential time (seconds)
Delaunay Refinement	1886.097	39532.22
Graph Coloring	1566.785	23214.378
Betweenness Centrality	25432.078	378835.65
K-means Clustering	16562.046	176791.708
PageRank	1678.381	48018.028
BlackScholes	25789.837	182441.231
Single Source Shortest Path	15067.332	197785.465

Table 3: Execution times at maximum speedup for the distributed execution configuration.

strates dyDSM's ability to handle parallel applications running across multiple machines. For six out of seven benchmarks, we achieve highest speedups when eight computation threads are run on each machine. This shows that the dyDSM communication layer utilizes the cores only to a limited extent; thus leaving them free to be used by the computation threads.

5.2 Performance of the Adaptive Scheme

Benchmark	Adaptive / Static
Delaunay Refinement	1.10
Graph Coloring	0.66
Betweenness Centrality	1.04
K-means Clustering	1.04
PageRank	1.22
BlackScholes	1.19
Single Source Shortest Path	1.54

Table 4: Execution time: Adaptive vs. Ideal static scheme.

This section evaluates the performance of the dynamic scheme that adapts the number of communication daemons at runtime (described in Section 4). Table 4 shows the execution time of the adaptive scheme normalized to the ones of the ideal static scheme. For the adaptive scheme, we set the initial daemon number to four. For ideal static scheme, we found the number of daemons that gave the best performance. From the table, we can see that the adaptive scheme greatly improves the performance over the best static scheme for benchmark `graph coloring`; this is because the number of chosen daemons varies over the program lifetime and this is aided by the dynamic scheme. We observe that the performance of `SSSP` is most degraded: in the static scheme `SSSP` performs best at 1 communication thread. The dynamic scheme performs worse because it favors more communication threads and therefore rarely runs at the 1-communication thread configuration. For the remaining benchmarks, the adaptive scheme performs close to that of the static scheme. The slight degradation can be attributed to the overhead of adaptation: periodically measuring the length of queue, waking up more communication threads, etc.

6 Related Work

Many previous works have been done on DSM systems [5, 22, 23]. However, *these DSM systems were not developed for modern clusters, where each node has many processing cores.* They do not make use of the multiple cores on each node. In addition, these DSM systems work poorly

for applications that use using dynamic features provided by modern programming languages. The performance of SPMD-based DSM is greatly degraded for modern applications as these dynamic features make the appropriate distribution of data impossible to determine at compile-time. In a cache-coherent DSM system, data needs to be replicated and/or migrated on-the-fly as it is impossible to statically figure out the memory access patterns for programs using dynamic features. dyDSM overcomes these drawbacks. We allow automatic data distribution, while keeping the communication overhead off the critical path by exploiting the multiple cores on each machine. Recently, software transactional memory has been introduced as a parallel programming paradigm for clusters [20, 3, 16, 17, 9]. Compared to them, we move a step further in hiding latency. Our communication mechanism makes efficient use of multicore machines. Latency tolerance is achieved by moving communication off the critical path, i.e., commits occur asynchronously in parallel with computations.

Partitioned global address space (PGAS) is a parallel programming model which tries to combine the performance advantage of MPI with the programmability of a shared-memory model. It creates a virtual shared memory space and maps a portion of it to each processor. PGAS provides support for exploiting data locality by affining each portion of the shared memory space to a particular thread. The PGAS programming model is used in UPC [8], Co-Array Fortran [11], Titanium [14], Chapel [6] and X10 [12]. UPC, Co-Array Fortran, and Titanium use SPMD style with improved programmability. Chapel and X10 extend PGAS to allow each node to execute multiple tasks from a task pool and invoke work on other nodes. These PGAS programming models primarily explore parallelism for array-based data-parallel programs using data partitioning. For the most part they do not consider dynamic data structures.

Given this works' focus on speculative parallelism for applications with low, non-deterministic sharing, we do not expect much benefit from sophisticated load balancing schemes as proposed in [25]. Moreover, at today's scale, such schemes can easily overwhelm the network. For purposes of this work, Memcached's data partitioning along with on-demand prefetching suffice.

7 Conclusion

This paper presented dyDSM, which is a DSM system designed for modern dynamic applications and clusters composed of multicore machines. It provides a simple to use programming interface and a powerful runtime and compiler that handles of the tedious tasks of using distributed-memory. dyDSM's support for large dynamic data structures makes it unique and very relevant for modern applications. The communication layer of dyDSM runtime makes use of multiple cores on machines and prefetching to high network latency. dyDSM also offers sequential consistency at coarse-grain level and supports speculative parallelism. Our evaluation of dyDSM on a cluster of five eight-core machines shows that it performs well giving an average speedup of 7x across seven benchmark programs.

References

[1] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *TSE*, 18:190–205, 1992.

[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, pages 72–81, 2008.

[3] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP*, pages 247–258, 2008.

[4] Erik B. Boman, Doruk Bozdağ, Unit Catalyurek, Assefaw H. Gebremedhin, and Fredrik Manne. A scalable parallel graph coloring algorithm for distributed memory computers. In *EURO-PAR*, pages 241–251, 2005.

[5] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *J. Supercomputing*, 2(2):151–169, 1988.

[6] Bradford Chamberlain, David Callahan, and Hans Zima. Parallel programmability and the chapel language. *IJHPCA*, 21(3):291–312, 2007.

[7] Jeffrey S. Chase, Darrell C. Anderson, Andrew J. Gallatin, Alvin R. Lebeck, and Kenneth G. Yocum. Network i/o with trapeze. In *HOTI*, 1999.

[8] UPC Consortium. UPC language specifications, v1.2. *Lawrence Berkeley National Lab Tech Report LBNL-59208*, 2005.

[9] Alokika Dash and Brian Demsky. Integrating caching and prefetching mechanisms in a distributed transactional memory. *TPDS*, 22(8):1284–1298, 2011.

[10] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Workshop on Large-Scale Parallel KDD Systems*, 1999.

[11] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A multi-platform co-array fortran compiler. In *PACT*, 2004.

[12] Philippe Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[13] Min Feng, Rajiv Gupta, and Yi Hu. SpiceC: scalable parallelism via implicit copying and explicit commit. In *PPoPP*, pages 69–80, 2011.

[14] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su, and K. Yelick. Titanium language reference manual. *U.C. Berkeley Tech Report, UCB/EECS-2005-15*, 2005.

[15] E Jul, H Levy, N Hutchinson, and A Black. An object-oriented language and system that indirectly supports dsm through object mobility. 1988.

[16] Christos Kotselidis, Mohammad Ansari, Kimberly Jarvis, Mikel Lujan, Chris Kirkham, and Ian Watson. DiSTM: A software transactional memory framework for clusters. In *ICPP*, pages 51–58, 2008.

[17] Christos Kotselidis, Mohammad Ansari, Kimberly Jarvis, Mikel Lujan, Chris Kirkham, and Ian Watson. Investigating software transactional memory on clusters. In *IPDPS*, pages 1–6, 2008.

[18] Milind Kulkarni, Martin Burtscher, Calin Casaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, 2009.

[19] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[20] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP*, pages 198–208, 2006.

[21] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Concurrency*, 4(2):63–79, 1996.

[22] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. In *ASPLOS*, pages 174–185, 1996.

[23] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *ASPLOS*, pages 297–306, 1994.

[24] Paul E. Utgoff, Neil C. Berkman, and Jeffery A. Clouse. Decision tree induction based on efficient tree restructuring. *Mach. Learn.*, 29(1):5–44, 1997.

[25] Robert P Weaver and Robert B Schnabel. Automatic mapping and load balancing of pointer-based dynamic data structures on distributed memory machines. In *Scalable High Performance Computing Conference, 1992. SHPCC-92. Proceedings.*, pages 252–259. IEEE, 1992.

[26] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for speculative parallelization of partially-parallel loops in dsm multiprocessors. In *HPCA*, pages 135–139, 1999.