UNIVERSITY OF CALIFORNIA
RIVERSIDE

Size Oblivious Programming of Clusters for Irregular Parallelism

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Sai Charan Koduru

December 2015

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Laxmi N. Bhuyan
Dr. Iulian Neamtiu
Dr. Evangelos Christidis

The Dissertation of Sai Charan Koduru is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

This dissertation is the fruition of all the support, kindness and inspiration I've received.

I am forever grateful and indebted to my advisor, Dr. Rajiv Gupta. Words cannot sufficiently express my thankfulness for his unwavering, constant and consistent confidence and belief in my potential and capabilities. For his inspiration and guidance in my research, writing and presentations. For his understanding and patience through my highs and my lows. Without his help, support, motivation and training, this would not be a reality. Thank you, Dr. Gupta, for the rewarding, memorable and cherishable doctoral study under your tutelage. And, thank you, Dr. Panchanathan, for kindly introducing me to Dr. Gupta.

Thank you, Dr. Neamtiu, for the collaboration over the years, guidance and for agreeing to be on my committee. I'd like to thank my dissertation committee members for their valuable feedback and support over the years. Thank you, Dr. Bhuyan and Dr. Hristidis. I'd also like to thank Dr. Morikis, for being on my advancement committee. And, thank you, Amy and Victor – you are the best!

Thank you, Kitchu and Saurabh, my life's sounding boards! A big, heartfelt *Thank You* to my buddies, lab mates and collaborators – Min, Kishore, Yan, Pamela, Changhui, Amlan, Vineet, Keval, Farzad, Zack, Bo, Prerna, Mehmet, Panruo, Hemanth, Prashanth and Vamsee, for the deep and meandering ramblings on research, life and everything in between.

My teachers from my primary school at Prasanthi Nilayam, specially Munni aunty and Prema Aunty – I am forever indebted to all of you for the loving care and being my family growing up in the hostel; you made it very hard for my parents to fill your shoes! Heartfelt gratitude to all my high school teachers, particular Sairam sir, for molding me and for giving me a hand and direction when I faltered. To Mishra sir, for the wonderful English and Philosophy classes. To Prusty sir, Satish Babu sir, Ram Mohan sir, Venkateshwarlu sir and Janardhanan sir for everything. To all my teachers in Bridavan campus – particularly, Subramanian sir – thank you for trusting a kid to tinker with computers – I learnt a lot fiddling with the machines which kindled the start of a long lasting love for computers and their science and to Warden sir for cutting me slack when I was down. Finally, a *HUGE* Thank You! to all my teachers, lecturers and professors at SSSIHL, Prasanthi Nilayam for my the formative years in Math and Computer Science. Thank you, Prof. GV Prabhakar Rao and Dr. Chase – my masters' thesis advisors and mentors. Thank you JB sir, Raghu sir, Pallav sir, Ravi Iyer sir and Uday sirs – and everyone else at DMACS, SSSIHL for believing in all of us kids! I hope I've lived up to at least a bit of all your collective expectations.

Thank you, Hari and Swathi for *everything* from day One! I'm forever indebted and grateful to my Mum, Dad for their love and constant prayers. I'm still stumped by my in-laws for nonchalantly trusting me with their little girl – thank you. Thank you, Chinna, Praveen and families – for bearing with my idiosyncrasies. And, none of this would have been possible without the unconditional love, co-operation and belief from my better-half, *Prasanthi*, for understanding and putting up with me through my grad-student life of classes, publications, accepts, rejects, missing out on family and the many, *many* little pleasures. I hope to make it up to you!

To my Guru, Guide and God, *Sri Sathya Sai*, for who I am.

To my wife, *Prasanthi*, for all the love and support.

ABSTRACT OF THE DISSERTATION

Size Oblivious Programming of Clusters for Irregular Parallelism

by

Sai Charan Koduru

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2015
Dr. Rajiv Gupta, Chairperson

Ubiquitous availability of growing troves of interesting datasets warrants a rewrite of existing programs, for clusters or for out-of-core versions, to handle larger datasets. While DSM clusters deliver programmability and performance via shared-memory programming and tolerating latencies by prefetching and caching, copious disk space is far more readily available than managing clusters. Irregular applications, however, are challenging to parallelize because the input related data dependences that manifest at runtime require use of *speculation* for correct parallel execution. By speculating that there are no input related cross iteration dependences, it can be `doall` parallelized; the absence of dependences is validated before committing the computed results. Latency tolerating mechanisms like prefetching and caching which have traditionally benefited data-parallel applications, can hurt performance of speculation from reading stale values. This thesis seeks to address the task of *size oblivious programming* of *irregular applications* for large inputs on DSM clusters.

We first simplify the task of programming very **large** and **irregular** data, which may sometimes not fit in the DSM. To this end, we introduce a language, the associated runtime and a compiler for efficient distributed software speculation to parallelize irregular applications. The programming model consists of an easy to use, templated C++ library for writing new vertex-centric programs or the simple `large` and `speculate` constructs as extensions to the C/C++ language for existing programs.

In addition we introduce the *InfiniMem* random-access efficient *object data format on disk* to enable I/O of arbitrary data objects from disk in *at most two logical disk seeks*. We also present a simple API for I/O into this data format and the accompanying object-centric library framework to program size oblivious programs to support scale-up on individual machines as a first step. We then leverage *InfiniMem* on individual machines in the cluster to support distributed size oblivious programs. As a final stage to ease programming, we built an LLVM/Clang-based source-to-source compiler, *SpeClang*, which instruments the annotated source code with invocations into our libraries and runtime. The runtime comprises of an object-based caching DSM with explicit support for software speculation and transparently performs out-of-core computations using *InfiniMem* to handle large inputs.

Next, we address the inefficiencies that result from employing traditional latency tolerance mechanisms like prefetching and caching on DSM systems to support speculation: (a) we demonstrate the need for balancing computation and communication for efficient distributed software speculation and provide an adaptive, dynamic solution that leverages prefetch and commit queue lengths to drive this balance. (b) we demonstrate that aggressive caching can hurt speculation performance and present three techniques to decrease communication and cost of misspeculation check and speed up misspeculated recomputations by leveraging the DSM caching and speculation protocols to keep misspeculation rates low.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The widespread availability of large scale clusters has enabled computer scientists to tackle applications that are both highly data-intensive and compute-intensive. However, for many of the modern workloads, exploiting the clusters requires a great deal of effort due to the need to handle *very large data sets* and exploit *irregular parallelism*. Examples of such workloads include graph processing on irregular inputs like social graphs etc., many of which are easily available to the research community [78, 72].

For ease of programming clusters the Distributed Shared Memory (DSM) systems have been proposed. For example, Orca [8, 7], Shasta [112, 113], Munin [10], TreadMarks [4], CRL [56]. In such systems the programmer develops the applications using the shared memory model which makes programming easy. The focus of these systems were latency tolerance mechanisms for DSMs like *prefetching* and *caching*. While great strides were made in developing distributed memory systems, these systems were not designed to either deal with very large data sets and nor were they developed to effectively exploit irregular parallelism. If the input data or intermediate data generated by an application is so large that it cannot be held in memory, these systems simply fail. The exploitation of irregular parallelism requires the use of speculation. However, much of the work on speculation is aimed at shared memory multicore systems [104, 105, 40, 99] and not applicable to distributed memory systems.

Existing programs that work perfectly well for smaller inputs, cannot run with larger inputs. Typically, the programmer will need to modify the program to explicitly handle larger inputs, for example, by using an *out-of-core* solution. Such manual, tailor-made solutions usually only work specific system configurations, constraints and assumptions and these programs simply crash when any of these assumptions are violated. For example, programs written using the popular Hadoop [118] framework require the programmers to *manually* ensure that the data being loaded into the java process can fit in the allocated JVM memory. Minor changes to cluster configurations, like lowering the per-process JVM memory can result in the program crashing. Other big data systems like Hive [126] focused on distributed relational queries using Hadoop, while dedicated, specialized systems like GraphX [131] and GraphLab [82] focused on distributed graph processing only. GraphChi [75] is one system that provides a size oblivious programming experience, but only for graphs and only works on with a single machine. In summary, although existing big data processing systems make use of disk in addition to memory, decisions made by the user cause the programs to be inflexible and they often fail because they run out of memory; further, they do not support speculation which is necessary for many irregular parallelism and/or they are usually domain specific solutions.

We therefore address the need for ***Size Oblivious Programming*** – techniques and runtime that can *transparently and efficiently* handle very large data, with minimal to no manual intervention. This dissertation develops a programming system that provides an easy to use programming interface for developing applications that must handle large data sets and contain irregular parallelism. The task of application developer is greatly simplified because the complexities of handling large data sets and irregular parallelism are handled transparently by the runtime system.

1.1 Dissertation Overview

This dissertation presents a parallel programming system shown in Figure 1.1 which addresses the issues of programmability and performance described above via design of a new programming interface and a new distributed shared memory system. The programming interface allows user to identify collections of objects that can become too large in number to be held in memory and indicate when the parallelism present is of speculative nature. The *InfiniMem* library augments distributed shared memory with a transparent use of disk to handle large collections of objects locally on each machine. The runtime distributed caching protocols on which the distributed shared memory system is based are specially designed to handle speculation.

Consider graphcoloring as an example application. The first phase is reading the input. If the input is very large and cannot entirely fit in memory, the program cannot progress any further and typically crashes with an out-of-memory error. In such a scenario, the application cannot progress any further. When the input data can fit in the available

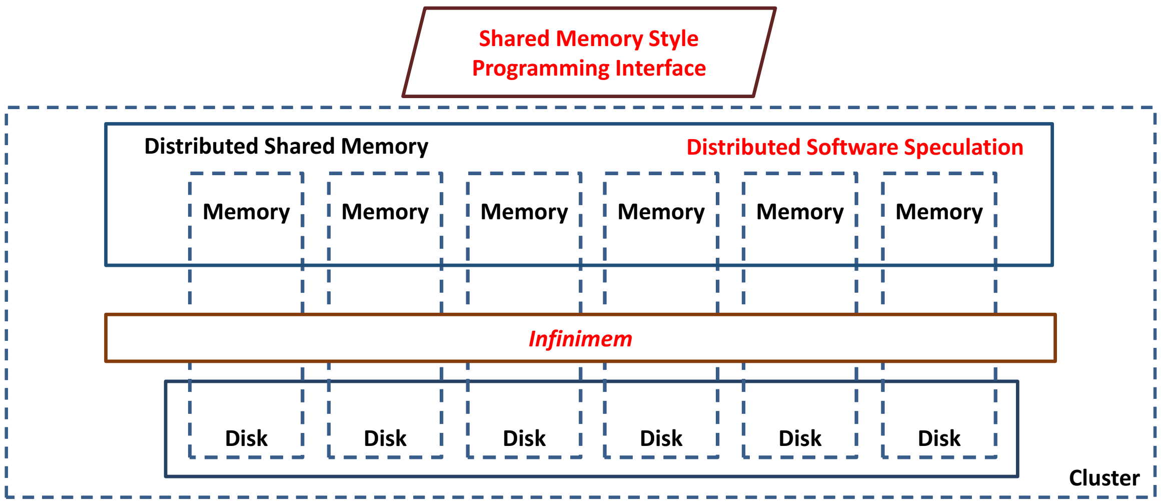


Figure 1.1: Overview of our Size Oblivious Programming approach.

memory, parallelizing an application like graph coloring requires the speculation, a runtime technique, since the dependencies between concurrent iterations are input dependent and are known only at runtime: to color nodes in a graph, the coloring algorithm needs to iterate over all nodes in the graph, comparing the color of each node with that of all of its neighbors. When naively parallelized, concurrently updating a graph node and one or more of its neighbors neighbors will result in incorrect coloring. Speculation avoids such mishaps. This dissertation addresses the problem of simplifying programming applications that process very large inputs and require speculation for correct execution on a cluster with distributed shared memory. We present a simple language and a runtime to aid with this.

1.1.1 Programming Interface

We first illustrate the proposed programming interface with the Graphcoloring example. We first address the problem of large input I/O and then address the problem of processing large irregular inputs.

Language Support for Large Inputs

The input to the graph algorithms can be any large input graph. If the programmer implicitly assumes that the input can fit in memory when the input graph cannot actually fit in the available memory, the program will crash when loading the graph after failing to allocate additional memory. We propose to transparently and automatically spill data that cannot fit in the available memory to disk. To enable this, we require the programmer to simply identify the large data collections to the framework with the introduction of the `large` keyword for container data structures that are potentially large. On the cluster, container data structures annotated with the `large` keyword are preprocessed to enable partitioning across machines in the cluster and then loaded into distributed memory in parallel for efficiency and performance. When the data does not fit in the available memory, the objects in the large collections are automatically and transparently spilled to and fetched from disk as needed by the application. Reads and writes to `large` data structure are transparently handled by the runtime library.

Language Support for Speculation

The sequential Graphcoloring example colors all nodes in the input graph in a loop, coloring nodes after considering the colors of all its neighboring nodes. Parallelizing this loop requires speculation since the list of node's neighbors can only be known at runtime. Speculation involves treating this loop as a *doall* loop and executing it in parallel in a distributed manner on the DSM system. When the nodes being processed in parallel are not neighbors, no misspeculation is encountered. However, if neighbors are processed in parallel, they may be assigned the same color leading to misspeculation. A commit phase that updates the colors of vertices detects this misspeculation and initiates recovery. The recovery requires repeating the coloring process for a vertex. If misspeculation occurs infrequently, speculative parallelization results in speedups. Speculation is expressed by using the `speculate` keyword. Distributed objects in the `large` container that are accessed inside the `speculate` block/method are automatically copied into a thread-private speculation buffer to enable the speculative computation to commit atomically.

1.1.2 Speculation on Shared-Memory Clusters

Irregular applications, such as vertex centric graph processing algorithms, contain significant levels of data parallelism as the same operation needs to be performed repeatedly over the large number of vertices in the graph. On a distributed system, the vertices and their processing can be distributed across the machines to exploit data parallelism. While use of DSM makes the task of programming easy and the distributed caching protocol reduces long latency fetch operations, the irregular nature of parallelism makes it necessary to employ speculation. Infrequently arising data dependences requires processing of interdependent vertices to be serialized. While these dependences are infrequent, they are dependent upon the input data set, and hence are revealed at runtime. Speculation [105, 104, 40] involves processing vertices in parallel assuming that there are no dependences between them, runtime detection of misspeculation when the dependences arise, and then recovering from them. As long as misspeculation rate is low, the benefits of speculation far outweigh the cost of recovering from misspeculation. Data parallel applications on DSMs benefits from prevailing techniques like prefetching and caching to tolerate the long network latencies for remote I/O.

However, aggressive prefetching and over-reliance on caching tend to increase misspeculation rates on DSMs. In this context, these are the problems that we tackle in this thesis:

1. Aggressive prefetching can increase misspeculation rates from prematurely fetching values that could become stale before the computation can commit.

2. For speculation execution, every read from the cache is a potential misspeculation and larger caches only increase the misspeculation rates.

Balancing Prefetching and Commit

The biggest challenge to efficiency in a distributed setup is the *large network latency* which comes into play for all remote `fetch`, `store` and `lock` operations. Prefetching and caching as techniques to ***tolerate long network latencies*** have been effective for general purpose DSM applications. However, in the context of speculation, overaggressive prefetching and caching can hurt the performance of speculation: computations that speculate on prefetched stale values *will* eventually abort, increasing the *misspeculation rate* and resulting in an overall slow down of the program execution. Therefore, it is imperative that the rate of prefetching should be paced with the speed of computation. To enable such balanced pacing, our computation model designates various types of threads: `prefetch`, `compute` and `commit` threads designated to their named tasks. To pace the prefetching, the length of the `commit` and `prefetch` queues are used as hints to auto tune the balance between `prefetch`, `compute` and `commit` to decrease over-reliance on prefetching and prioritizing commits.

Decreasing Misspeculation Rate with Caching DSMs

Another challenge in this regard is the presence of machine level caches in DSMs. While DSMs coupled with caching help tolerate long network latencies for non-speculative data-parallel applications, aggressive caching can hurt during speculative execution due to the potential of using stale values in computations, increasing the *misspeculation rate*. With distributed speculation on a caching DSM, every cache read can cause misspeculation because the read could occur before a network-delayed invalidate is received – note that successful speculation is counting on the fact that no invalidate would arrive. To address this challenge, we develop optimizations for distributed speculation on caching based DSMs that decrease the cost of misspeculation check and speed up the re-execution of misspeculated recomputations.

1.1.3 Efficient I/O for Large Irregular Inputs

When the input data is very large and does not fit in the DSM memory, we propose that the data that does not fit in memory can be *transparently* offloaded to the disk and fetched into memory on demand. This is akin to an out-of-core solution. While many out-of-core algorithms have been proposed, but we go a step further and propose generic techiques to transparently compute *any* algorithm out-of-core.

A naive apporach to this would be to use a database to spill and fetch the overflow data, we show that this approach is not very efficient for irregular data due to the large number of unpredictable random accessess generated by applications processing irregular inputs. For example, we experimented with RocksDB [50], which is an enhancement of LevelDB [51]. RocksDB claims to have better support for random reads and graph scans; however, we show that we can do far better for irregular applications. For example, GraphColoring on the Amazon [77] input with 400K vertices and 3,200K edges took about 23 seconds with RocksDB as the excess datastore while it only took about 1 second with our solution, *InfiniMem*.

Application	Load/Store in seconds (% of total)	Compute in seconds (% of total)	RocksDB I/O in seconds (% of total)	Total Time in seconds
Coloring	9 (2.56)	7 (2.01)	333 (95.43)	349
Conn. Comp.	15 (2.50)	10 (1.67)	574 (95.83)	599
Comm. Dete.	13 (1.95)	20 (3.00)	633 (95.05)	666
SSSP	14 (2.06)	13 (1.92)	650 (96.02)	677
PageRank	10 (1.01)	21 (2.12)	955 (96.87)	986

Table 1.1: I/O costs are significant in the total running time for external irregular algorithms.

Table 1.1 shows the time breakdown for loading, storing, spilling and computation for various applications on the Amazon [77] input using RocksDB as the datastore as described above. The program was configured to hold upto 20,000 vertices in memory while performing the computation, spilling the remaining vertices to disk and loading them on demand as needed. We see that the time spent in I/O is a significant portion of the total running time. To begin, for SSSP and Conn. Comp., the sequential load/store times are more than the time spent on computation. And, the time spent by these benchmarks on compute is an *insignificant* portion of the total time: only ∼2% is spent on compute and input load/output store each. Instead, much of the total running time is spent in the RocksDB I/O: ∼95% is spent on intermediate I/O with RocksDB, motivating the need for efficiencies across the board, including:

1. Parallel loading and storing of inputs and outputs;

2. Need for speculation to parallelize irregular applications; and,

3. Need for efficient datastore for excess data with support for fast random access.

Since the data access patterns are irregular, they result in unpredictable random accesses and the resulting spilling and loading data in contiguous chunks would be inefficient with naive solutions. Instead, we need the ability to *transparently and efficiently* spill and load individual objects to and from disk without the traditional penalties associated with random seeks on a disk. We propose a data representation on disk that allows efficient *random access* to spill and retrieve objects of irregular inputs to and from disk. Specifically, with *InfiniMem* we provide a runtime and API to efficiently spill and access irregular data on disk.

1.2 Dissertation Organization

The rest of the thesis is organized as follows: Chapter 2 presents the language to support `Large` inputs and to program distributed software `speculation`. Next, Chapter 3, presents *InfiniMem*, our out-of-core solution to *transparently* scale computation to inputs that do not fit in the available memory. Distributed software speculation is convered in the next two chapters: Chapter 4 addresses the need to balance prefetching and commit for lowering misspeculation rates while Chapter 5 addresses the problem of optimizing distributed speculation in the presence of caching on DSM clusters. Chapter 6 surveys existing literature in related areas and Chapter 7 concludes the thesis with a summary of our work and presents a brief future outlook.

Chapter 2

Programming Interface for Size Oblivious Programming

The use of Distributed Shared Memory (DSM) systems is a widespread in modern computing. While early systems relied on custom message passing solutions via `send` and `recv` primitives, the introduction of the Message Passing Interface (MPI) [44] standard and accompanying implementations like OpenMPI [93], MPICH [48] etc. eased the programming of these systems. The runtime libraries managed efficient packing and network based I/O of any kind of data structures that can be expressed as an `MPI_Type_struct`. Systems like Orca [8], Shasta [112], Munin [10], TreadMarks [4], CRL [56] etc. implemented distributed caching protocols with runtimes invoking message passing to implement the caching protocols. The next phase of simplification involves utilizing the compiler to generate calls into MPI or similar libraries, thereby simplifying DSM programming even further. Examples include Chapel [16], X10 [25], UPC [39], SHMEM [24] etc. However, these prior systems did not have explicit support for today's emerging workloads that require *speculative parallelism* nor did they provide very simple language constructs for *large* distributed data collections.

This chapter presents the two simple language extensions to the C++ language to support size-oblivious irregular programs. We then present examples illustrating the use of these constructs in the context of irregular graph applications. Next, we show the transformation of programs written using these constructs are transformed into their *object-centric* counterparts. And finally, we present the low-level language and API that powers the runtime and describe how these can be customized for easily evaluating different algorithms and/or strategies etc.

2.1 High-level Language for Size-Oblivious Irregular Parallelism

A lot of emerging applications involve distributed parallel processing of irregular data structures like large graphs and networks. Graphs are essentially *collections* of vertices and edges. Large graphs have a large number of vertices and/or edges. It would be convenient for the programmer to simply identify such collections and have the language and/or runtime handle the detailed nitty-gritties of distributing and managing the collection. We wish to provide a simple language keyword that enables this simplicity:

```
large;
```

Parallelizing irregular applications requires the use *speculation* for correct parallel execution. While software speculation has been extensively researched on single shared memory systems, speculation in a distributed setup is still work in progress. For example, the LRPD test [105] formalized the notion and procedure for safe speculative execution via privatization. SpiceC [40] introduced the `#pragma speculate` language construct to speculatively parallelize loops on single shared memory sytems. We wish to introduce a similar simple language keyword that enables speculative parallelization in a distributed setup:

```
speculate{ ... };
```

We now illustrate the use of `large` and `speculate` with the example of sequentially loading and coloring a potentially large graph. This example is listed in Algorithm 4.1, which presents *two* opportunities for parallelization and speedup: the graph loading loop on Line 1 can be parallelized and the graph coloring loop on Line 3 can also be parallelized.

Algorithm 1: Sequential graph loading and the graphcoloring algorithm.

```
1  Vertex graph[NUM_VERTICES];
2  /* Read edgelists and add neighbors */
3  while moreInputEdges() do
4      readNextEdge(src, dst);
5      graph[src].addNeighbor(dst);
6  /* Graphcoloring */
7  for int i=0; i<graph.size(); i++ do
8      for int j=0; j<graph[i].numNeighbors(); j++ do
9          if graph[i].nbr[j].color == graph[i].color then
10             graph[i].color++;
```

Parallelizing the graph input loop on Line 3 requires the use of speculation since reading edges in parallel could result in multiple neighbors concurrently added to the same vertex. In order to not lose neighbors added in this process, we require speculation. As before, the input graph itself could be very large and automated memory and I/O management for this data would be desirable.

Next, parallelizing the graph coloring loop on Line 2 also requires speculation since a naive `doall` parallelization of the loop could result in a vertex and its neighbor concurrently processed, leading to incorrect coloring. Further, the input graph on Line 1 could potentially be an input that does not fit in available memory, and could benefit from automatic memory and storage management by spilling from memory to disk and fetching from disk into memory automatically and transparently.

The simple annotations to transform this program into a size-oblivious program with distributed software speculation are shown in Algorithm 2. The `large` keyword on Line 1 and the `speculate` blocks on Line 3 and Line 9 are the only required changes to achieve this.

Algorithm 2: Use of `large` and `speculate` for parallel loading and processing of large, irregular inputs.

```
1  large Vertex graph[NUM_VERTICES];
2  /* Read edgelists and add neighbors */
3  speculate {
4  while moreInputEdges() do
5      readNextEdge(src, dst);
6      graph[src].addNeighbor(dst);
7  }
8  /* Graphcoloring */
9  speculate {
10 for int i=0; i<graph.size(); i++ do
11     for int j=0; j<graph[i].numNeighbors(); j++ do
12         if graph[i].nbr[j].color == graph[i].color then
13             graph[i].color++;
14 }
```

2.2 Low-level API for Size-Oblivious Irregular Parallelism

The high-level `large` and `speculate` language extensions are easy to use in a everyday programming. However, the bulk of the heavy lifting to enable the semantics of these constructs is handled by the acommpanying runtime and libraries. Our library provides a rich set of low-levl APIs to handle the nittigritties of size-oblivious programming and speculation.

2.2.1 Object-centric Programming

Typical algorithms for big data processing usually iterate over all input data items. So, the high-level design of the language is to transform the `large` and `speculate` constructs into their corresponding *Object-centric* versions, where the transformed program also iterates over all the objects in the input data. We illustrate this with an Graph Coloring as a typical example. Modern graph-processing frameworks are so-called *vertex-centric* frameworks, where the unit of work is around individual vertices. Object-centric versions are the equivalent generalizations for arbitrary datatypes where the program iterates over all objects in the container.

We first illustrate the transformation of an annotated program into its equivalent object/vertex-centric versions, followed by a summary of all the available low-level APIs that are automatically invoked by the runtime to process the inputs speculatively and in parallel.

Figure 2.1 illustrates the transformation of a program annotated with the `large` and `speculate` C++ extensions as seen in Figure 2.1a into a vertex-centric parallel version in Figure 2.1b. These extensions are encapsulated in the *SpeClang* complier and associated runtime. Observe the transformation of the `large` keyword into the `Large<>` template container with the same datatype on Line 1. Further observe that the `speculate` block is transformed into the body of the `Large::Speculate() method`.

```
 1 large Vertex g[NUM_VERTICES];
 2
 3 int main() {
 4  load(g, ‘data’);
 5
 6
 7  for(int i=0; i<NUM_VERTICES; i++)
 8   speculate {
 9    for(int j=0; j<g[i].nbrs(); j++)
10     if(g[i].color == g[i].nbrs[j].color)
11      g[i].color++;
12    }
13  return 0;
14 }
```

(a) Program annotation using *SpeClang*.

```
Large<Vertex> g;

template <typename V>
Large::Speculate(V v) {
  for(auto nbr: v) {
    if(v.color == nbr.color)
      v.color++;
  }
}

int main() {
  g.preprocess(‘input’).load().run();
  return 0;
}
```

(b) Object-centric speculative programming.

Figure 2.1: Object-centric programming and annotation of speculative regions with *SpeClang*.

2.2.2 Low-level API

The high-level language presented above eases everyday programming. However, for purposes of research, exploration and customization to specialized domains, it would be convenient to easily tweak portions of the programming model. This is enabled by the low-level API.

We now describe the various portions of the low-level API that enables size-oblivious irregular parallelization. These include object identification, partitioning, I/O, processing, versioning and object locking. We bundled all these features into an object-based, Distribued Shared Memory (DSM) system with a directory-based caching protocol with strict consistency [129]. As illustrated in Figure 2.1, the high-level program annotations are transformed into the object-centric versions. The runtime that executes the transformed program on the DSM system which satisfied object fetches and stores through background calls to the low-level APIs.

Object Identifiers

The first step in enabling size-oblivious programming is a generalization of the notion of a pointer. We need a construct that is unique and portable across the entire distributed shared memory address space. In addition, this construct should also help identify/locate objects quickly and uniformly from any machine in the cluster. For this, we use the notion of an object identifier, or `ID`.

When injesting input data, each dataitem is assigned a unique `ID`. Objects are then referenced by their `ID` in the rest of the program. For example, consider a graph input which consists of vertices and their edges. Vertices are typically indexed by a numeric identifier. In the simplest case, the numeric identifier is the Vertex `ID`. The transformation of the program into the object-centric version ensures that the `ID` is consistently used throughout the program.

Input Partitioning

Once injested, the input then partitioned across machines in the cluster and assigned a *home* node based on their `ID`. A consistent global partitioning funcion ensures that the runtime on any machine can easily locate the home node of objects and initiate various network-based or local interations as needed. We provide a `preprocess()` API that can be overloaded to implement any custom partitioning and/or object identifcation.

Network I/O

One of the primary features of a DSM system are transparent data caching and consistency protocols which are enabled by `fetch` and `store` APIs. Consistent with the object-centric programming model, our DSM system is an object-based system. The basic `fetch` and `store` APIs have object granularity. In addition, the DSM also supports *batched* I/O APIs to fetch or store. This API helps with decreasing network communication by grouping/bundling I/O of objects to the same machine.

Object Versioning

Software speculation proceeds in phases: (a) Privatization, (b) Parallel Speculative Computation, (c) Misspeculation detection/check, and (d) Commit. To enable the third phase, we employ versioning of data to detect mis-speculation. Therefore, in addition to an object `ID`, each object in the DSM system is associated with a `version`. Each time an object is successfully committed to the DSM system, its version number is also atomically and automatically incremented to reflect the change in value.

Object Locking for Speculation

For the first two phases of speculation, the network I/O API provided by the DSM are sufficient. However, mis-speculation detection/check requires explicit locking of possibly local and remote objects to enable the atomic `commit` protocol after a successful mis-speculation check. For this, the DSM system also provides the `lock` and `unlock` API at object granularity.

Parallel Speculative Computation

Once the input objects are identified, partitioned and distributed across machines in the cluster, the `run` API is used to invoke the speculative parallel execution. This API invokes the object-centric speculative computation, `Large::Speculate()` on each object in the input dataset then invokes the `commit` API to ensure correct commit.

Mis-speculation Detection and Commit

Finally, the misspeculation detction and commit protocols are encapsulated in the `commit` API. The `commit` API takes as input the speculative buffer with private copies of dependent data computed results. As part of the commit, the API first locks all the dependent inputs and then performs a version check as part of the misspeculation detection. In the absence of misspeculation, updated values are written to the home nodes; otherwise, the computation is discarded and rescheduled.

```
template<typename T>                    template<typename T>
bool Large::assignID();                 bool Large::preprocess();


template<typename T>                    template<typename T>
T Large::fetch(ID);                     void Large::store(ID, const T*);
template<typename T>                    template<typename T>
T* Large::fetchBatch(set<ID> ids);      void Large::storeBatch(set<ID> ids);


template<typename T>                    template<typename T>
bool Large::lock(ID);                   bool Large::unlock(ID);

template<typename T>                    template<typename T>
bool Large::run();                      bool Large::commit(specBuffer &sb);
```

Figure 2.2: Complete list of low-level APIs to support object-centric program transformation and protocol customization to identify, pre-process, fetch/store, lock/unlock, parallel compute and speculation.

Figure 2.2 lists the complete set of low-level APIs that are used to implement the object-centric size-oblivious parallel processing system for irregular inputs. Observe that the API is templated to allow for processing of arbitrary datatypes.

As stated earlier, the low-level API serves the dual role of providing the runtime and also enables advanced programmers and researchers experiment with variations and customizations to the various pieces of the runtime, starting from object identification and all the way up to the speculation protocol.

2.3 The *SpeClang* Compiler and Runtime

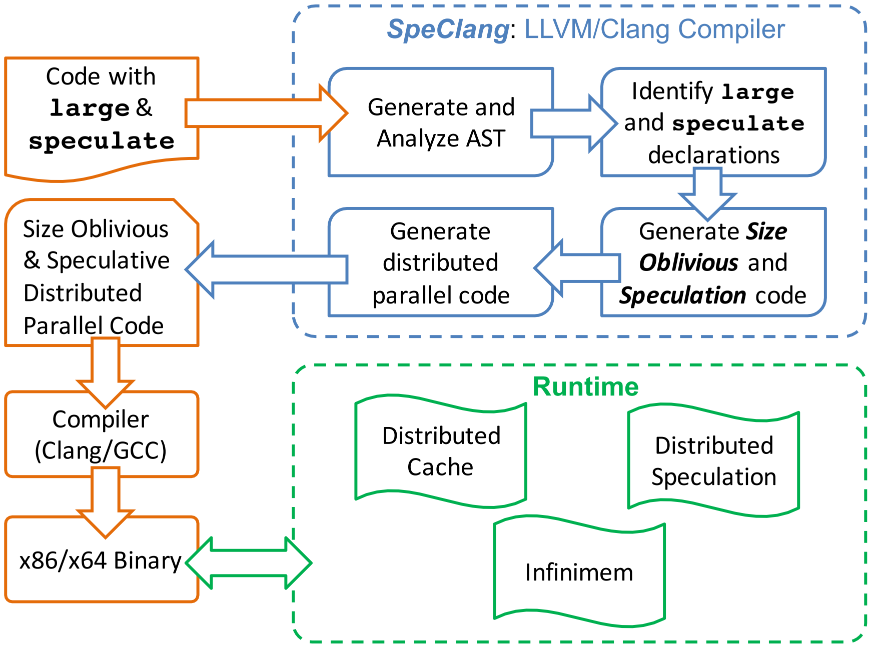

SpeClang: LLVM/Clang Compiler
Code with large & speculate
Generate and Analyze AST
Identify large and speculate declarations
Generate Size Oblivious and Speculation code
Generate distributed parallel code
Size Oblivious & Speculative Distributed Parallel Code
Compiler (Clang/GCC)
x86/x64 Binary
Runtime
Distributed Cache
Distributed Speculation
Infinimem

Figure 2.3: Overview of the *SpeClang* compiler and runtime implementation.

An overview of *SpeClang* framework is shown in Figure 2.3. The user annotates the C/C++ source code with `large` and `speculate` directives which is then source-translated into a size oblivious, parallel and distributed version. This source code is then built with any C/C++ compiler and is run on the DSM with distributed speculation and *InfiniMem*.

2.4 Summary

In this chapter, we have presented the simple high-level language extensions to C++ to support size-oblivious irregular programs. Specifically, we have introduced the `large` and `speculate` keywords to annotate potentially large collections and code regions requiring speculation respectively. We also showed the transformation of such annotated programs into their object-centric versions. Finally we described the low-level API that powers the runtime to achieve the dual goal of distributed speculation with support for very large data. The low-level API is provided to allow researchers and other advanced users to tweak and customize the algorithms and design to suit their own problem domains, while still holding together the overall framework.

Next, Chapter 3 presents and evaluates extensions to the low-level language API and a corresponding data organization on disk that enables application scale-up on a single machine. In addition to block-based I/O, the extensions provide features that allow fast and efficient random I/O that is well suited for irregular applications. These extensions also continue the object-centric design theme.

Subsequent chapters evaluate the distributed speculation enabled by this language along with targeted enhancements and optimizations that boost the performance of speculation on DSM systems that employ caching.

Chapter 3

Infinimem: Size Oblivious Programs on a Single Machine

Supporting size-oblivious programs requires a graceful handling of situations when the data to be processed does not fit in available memory. In such situations, we would like to scale gracefully by relying on the underlying disk – we can intelligently spill overflow data from memory to disk and bring back required data to memory transparently, as needed by the application, enabling us to scale the application further even when the DSM memory is full. A step in this direction is the *InfiniMem* library, which enable application *scale-up* on a single machine, thereby enabling better realiability in the application *scale-out* scenario.

BigData processing frameworks are an important part of today's data science research and development. Much research has been devoted to scale-out performance via distributed processing [46, 82, 84, 118] and some recent research explores scale-up [5, 135, 75, 99, 117, 133]. However, these scale-up solutions typically assume that the input dataset fits in memory. When this assumption does not hold, they simply fail. For example, experiments by Bu et al. [17] show that different open-source Big Data computing systems like Giraph [5], Spark [133], and Mahout [125] often crash on various input graphs. Particularly, in one of the experiments, a 70GB web graph dataset was partitioned across 180 machines (each with 16 GB RAM) to perform the PageRank computation. However, all the systems crashed with `java.lang.OutOfMemoryError`, even though there was less than 500MB of data to be processed per machine. In our experiments we also found that GTgraph's popular R-MAT generator [6], a tool commonly used to generate power-law graphs, crashed immediately with a *Segmentation Fault* from memory allocation failure when we tried to generate a graph with 1M vertices and 400M edges on a machine with 8GB RAM.

Motivated by the above observations, in this chapter, we develop *InfiniMem*, a system that enables *Size Oblivious Programming* – the programmer develops the applications without concern for the input sizes involved and *InfiniMem* ensures that these applications do not run out of memory. Specifically, the *InfiniMem* library provides interfaces for transparently managing a large number of objects stored in files on disk. For efficiency, *InfiniMem* implements different read and write policies to handle objects that have different characteristics (fixed size vs. variable size) and require different handling strategies (sequential vs. random access I/O). We demonstrate the ease of programming with *InfiniMem* by programming BigData analysis applications like frequency estimation, exact membership query, and Bloom filters. We further demonstrate the *versatility* of *InfiniMem* by developing size oblivious graph processing frameworks with three different graph data representations: vertex data and edges in a single data structure; decoupled vertex data and edges; and the shard representation used by GraphChi [75]. One advantage of *InfiniMem* is that it allows researchers and programmers to easily experiment with different data representations with minimal additional programming effort. We evaluate various graph applications on various inputs with three different representations. For example, a quick and simple shard implementation of PageRank with *InfiniMem* achieves performance within ∼30% of GraphChi.

The remainder of the chapter is organized as follows: Section 3.1 motivates the problem and presents the requirements expected from a *size oblivious programming* system. Section 3.2 introduces the programming interface for size oblivious programming. Section 3.3 describes the object representation used by *InfiniMem* in detail. Section 3.4 describes the experimental setup and results of our evaluation and Section 3.5 summarizes the chapter.

3.1 Size Oblivious Programming

The need to program processing of very large data sets is fairly common today. Typically a processing task involves representing the data set as a large collection of objects and then performing analysis on them. When this large collection of objects does not fit in memory, the programmer must spend considerable effort on writing code to make use of disk storage to manage the large number of objects. In this work we free the programmer from this burden by developing a system that allows the programmer to write *size oblivious* programs, i.e., programs where the user need not explicitly deal with the complexity of using disk storage to manage large collections of objects that cannot be held in available memory. To enable the successful execution of size oblivious programs, we propose a general-purpose programming interface along with an *I/O efficient* representation of objects on disk. We now introduce a few motivating applications and identify requirements to achieve I/O efficiency for our *size oblivious programming* system.

3.1.1 Motivating Applications

Algorithm 3: Motivating applications: Membership Query, Mesh Generation and Graph Processing.

```
1  HashTable ht;
2  while read(value) do
3      ht.insert(value);
4  while more items do
5      if ht.find(item) then
6          print("Item found");
7  ------------------------------------------------
8  Mesh m(NUM-VERTICES)
9  foreach node n in Mesh m do
10     i ← rand(0, MAX);
11     for j=0; j < i; j++ do
12         n.addNeighbor(m[j]);
13         m[j].addNeighbor(n);
14 foreach Node n in Mesh m do
15     Write(n)
16 Graph g;
17 while not end of input file do
18     read next;
19     g.Add( α(next) );
20 repeat
21     termCondition ← true;
22     forall the Vertices v in g do
23         for int i=0; i<v.nbrs(); i++ do
24             Vertex n = v.neighbors[i];
25             if v.dst>n.dst+v.wt[i] then
26                 v.dst←(n.dst+v.wt[i]);
27     if NOT converged then
28         termCondition ← false;
29 until termCondition is true;
30 foreach Node n in Graph g do
31     Write(n);
```

Consider an application that is reading continuously streaming input into a Hash Table in heap memory (lines 1–3, Algorithm 3); a website analytics data stream is an excellent example of this scenario. When the memory gets full, the `insert` on line 3 could fail, resulting in an application failure. Consider a common approach to graph generation which assumes that the entire graph can be held in memory during generation, as illustrated by lines 8–15 in Algorithm 3. First, memory for `NUM-VERTICES` is allocated (line 8) and then the undirected edges are generated (lines 11-13). Note that the program can crash *as early as line 8* when memory allocation fails due to a large number of vertices. Finally, consider the problem of graph processing, using SSSP as a proxy for a large class of graph processing applications. Typically, such applications have three phases: (1) input, (2) compute, and (3) output. The pseudocode for SSSP is outlined in lines 16–31 in Algorithm 3, highlighting these three phases. *Note that if the input graph does not fit in memory, this program will not even begin execution.*

3.1.2 A Naive Approach

To study the state of the art in this regard, we evaluated the use of RocksDB [50] to write a size-oblivious program. For this, we wrote an *object-centric* program to process large inputs: the program reads in sufficient data to fit in memory, process it in parallel and store the contents back to disk, thereby making space in memory for the next batch of data. For this, we evaluated the Graph Coloring algorithm using RocksDB as the data store. LevelDB [51] is touted as an efficient data store for large-scale applications. However, we used RocksDB – an evolution of LevelDB, as the data store in our test application since it claims to better support random accesses and graph processing. However, GraphColoring on the Amazon [77] input with 400K vertices and 3,200K edges took about 23 seconds with RocksDB as the excess datastore while it only took about 1 second with our solution, *InfiniMem*, which we now describe. This is attributed to RocksDB's data organization that requires loading an complete level into memory and performing a linear scan on that level. RocksDB is better suited to *scans* once a level is loaded. Out solution is more efficient for random accesses and requires at most 2 logical seeks to access any data object, in addition to supporting efficient block-based I/O.

3.1.3 Our Solution

We focus on supporting size oblivious programming for C++ programs via the *InfiniMem* C++ library and runtime. Examples in Algorithm 3 indicate that the data structures that can grow very large are usually represented as *collections* of objects. Size Oblivious Programming with *InfiniMem* simply requires programmers to *identify potentially large collections of objects* using very simple abstractions provided by the library and these collections are transparently made *disk-resident* and can be efficiently and concurrent accessed. We now analyze these representative applications to tease out the requirements for *size oblivious programming* that have influenced the architecture of *InfiniMem*.

Let us reconsider the pseudocode in Algorithm 3, mindful of the requirement of efficient I/O. Lines 5–6 will execute for *every* key in the input; similarly, lines 9 and 14 indicate that lines 10–13 and line 15 will be executed for *every* node in the mesh. Similarly, line 22 indicates that lines 23–26 will be performed on *every* vertex in the graph. It is natural to read a contiguous *block* of data so that no additional I/O is required for lines 24–26 for the vertices and is an efficient disk I/O property. Moreover, this would be useful for any application in general, by way of decreasing I/O requests and batching as much I/O as possible. Therefore, we have our first requirement:

Support for efficient block-based IO.

Consider next, the example of the hash table where the input data is *not* sorted; then, line 3 of Algorithm 3 motivates need for random access for indexing into the hash table. As another example, observe that line 24 in Algorithm 3 fetches every neighbor of the current vertex. When part of this graph is disk-resident, we need a way of efficiently fetching the neighbors, much like *random access* in memory. This is important because any vertex in a graph serves two roles: (1) vertex and (2) neighbor. For the role (1), if vertices are contiguously stored on disk block-based I/O can be used. However, when the vertex is accessed as a neighbor, the neighbor could be stored anywhere on disk, and thus requires an imitation of random access on the disk. Hence our next requirement is:

Support for efficient, random access to data on disk.

To make the case for our final requirement, consider a typical definition of the HashTable shown in Figure 3.1a. Each `key` can store multiple values to support *chaining*. Clearly, each `HashTableEntry` is a variable sized entity, as it can hold multiple `values` by chaining. As another example, consider the definition for a *Vertex* shown in Figure 3.1b: the size of `neighbors` array can vary; and with the exception of the `neighbors` member, the size of a *Vertex* can be viewed as a fixed-size object. When reading/writing data from/to the disk, one can devise very fast block-based I/O for fixed-size data (Section 3.3). However, reading variable-sized data requires remembering the size of the data and performing n reads of appropriate sizes; this is particularly wasteful in terms of disk I/O bandwidth utilization. For example, if the average number of neighbors is 10, every time a distance value is needed, we will incur a 10x overhead in read but useless data. As a final example, Figure 3.1c is an example of an arbitrary container that showcases the need for both fixed and variable sized data. Hence we arrive at our final requirement from *InfiniMem*:

Support to speed up I/O for variable-sized data.

The goal of *InfiniMem* is to transparently support disk-resident versions of object collections so that they can grow to large sizes without causing programs to crash. *In-*

```
template <typename K, typename V>
struct HashTableEntry {
  K key;
  V* values; /* for chaining */
};
```

(a) Hash Table

```
struct Vertex {
   int distance;
   int* weights; /*Edge weights*/
   Vertex* neighbors;   /*Array*/
};
```

(b) Graph Vertex

```
template<typename T>
struct Container{
  T stackObjects[96]; /* Fixed */
  T *heapObjects;  /* Variable */
};
```

(c) Arbitrary container

Figure 3.1: Common data structure declarations to motivate the need for explicit support for fixed and variable sized data, block based and random IO.

finiMem's design allows size oblivious programming with little effort as the programmer merely identifies the presence and processing of potentially large object collections via *InfiniMem*'s simple programming interface. The details of how *InfiniMem* manages I/O (i.e., uses block-based I/O, random access I/O, and I/O for fixed and variable sized data) during processing of a disk-resident data structure are hidden from the programmer.

3.2 The *InfiniMem* Programming Interface

In order to work consistently with our object-based DSM, *InfiniMem* uses a similar templatized interface as the DSM, with extensions to support efficient disk operations on large irregular inputs.

InfiniMem is a C++ template library that allows programmers to identify size oblivious versions of fixed- and variable-sized data collections and enables transparent, efficient processing of these collections. We now describe *InfiniMem*'s simple application programming interface (API) that powers *size oblivious programming.* *InfiniMem* provides a *high-level* API with a default *processing* strategy that hides I/O details from the programmer; however the programmer has the flexibility to use the *low-level* API to implement any customized processing.

3.2.1 Identifying Large Collection of Objects

In *InfiniMem*, the programmer identifies object collections that potentially grow large and need to be made disk-resident. In addition, the programmer classifies them as fixed or variable sized. This is achieved by using the `Box` and `Bag` abstractions respectively.

```
template<typename T>
struct Container: public Box<T> {//Bag<T>
  T data;
  void update() {  /* for each T */
     ...
  }

  void process();
};

typedef Container<int> intData;

typedef Container<MyObject> objData;

int main() {
   Infinimem<intData> idata;
   idata.read("/input/file");
   idata.process();

   Infinimem<objData> odata;
   odata.read("/input/data/");
   odata.process();
}
```

```
template<typename T>
T Box::fetch(ID);

template<typename T>
T* Box::fetchBatch(ID, count);

template<typename T>
void Box::store(ID, const T*);

template<typename T>
void Box::storeBatch(ID, count);

template<typename T>
T Bag::fetch(ID);

template<typename T>
T* Bag::fetchBatch(ID, count);

template<typename T>
void Bag::store(ID, const T*);

template<typename T>
void Bag::storeBatch(ID, count);
```

Figure 3.2: Programming with *InfiniMem*: the `Box` and `Bag` interfaces are used for *fixed size* and *variable sized* objects; `process` drives the computation using the user-defined `update()` methods and the low-level `fetch()` and `store()` API.

The `Box` abstraction can be used to hold fixed-size data, while the `Bag` holds flexible-sized data. Figure 3.2 illustrates an example and lists the interface. The programmer uses the `Box` or `Bag` interface by simply inheriting from the `Box` (or `Bag`) type and provides an implementation for the `update()` method to process each object in the container. Here, `Container` is the collection that can potentially grow large, as identified by the programmer. *InfiniMem*'s `process()` function hides the details of I/O and fetches objects as needed by the `update()` method, thereby providing a *size oblivious programming* experience.

3.2.2 Processing Data

The `process()` method is the execution engine: it implements the low-level details of efficiently fetching objects from the disk, applies the user-defined `update()` method and efficiently spills the updated objects to disk. Figure 3.3 details the default `process()`. By default, the `process()`-ing engine `fetch`es, `processes` and `store`-es data in batches of size `BATCH_SIZE` such that the entire batch fits and can be processed in memory.

```
// SZ = SIZEOF_INPUT; BSZ = BATCH_SIZE;
Box<T>::process() { // or Bag<T>
  for(i=0; i<SZ; i+=BSZ) {
    // fetch a portion of Box<T> or Bag<T>
    cache = fetchBatch(ID(i), BSZ);
    for(j=0; j<BSZ; j++)
      cache[j].update();
  }
}
```

Figure 3.3: *InfiniMem*'s generic batch `process()`-ing.

While *InfiniMem* provides the default implementation for `process()` shown in Figure 3.3, this method can be overridden: programmers can use the accessors and mutators exposed by *InfiniMem* (Figure 3.2) to write their own processing frameworks. Notice that *InfiniMem* natively supports both sequential/block-based and random accessors and mutators, satisfying each of the requirements formulated earlier. For block-based and random access, *InfiniMem* provides the following intuitively named fetch and store APIs: `fetch()` and `fetchBatch()`; and `store()` and `storeBatch()`.

Illustration of *InfiniMem* for Graph Processing

We next demonstrate *InfiniMem*'s versatility and ease of use by programming graph applications using *three* different graph representations. We start with the standard declaration of a `Vertex` as seen in Figure 3.1b. An alternate definition of `Vertex` separates the fixed sized data from variable sized edgelist for IO efficiency, and used in many vertex centric frameworks [82, 75]. Finally, we program GraphChi's [75] shards.

Figure 3.4a declares the `Graph` to be a `Bag` of vertices, using the declaration from Figure 3.1b. With this declaration, the programmer has identified that the collection of vertices is the potentially large collection that can benefit from size oblivious programming. The `preprocess()` phase partitions the input graph into disjoint intervals of vertices to allow for parallel processing. These examples uses a vertex-centric graph processing approach where the `update()` method of `Vertex` defines the algorithm to process each vertex in the graph. The `process()` method of `Graph` uses the accessors and mutators from Figure 3.2 to provide a *size oblivious programming* experience to the programmer. Figure 3.4b declares a `Graph` as the composition of a `Box` of `Vertex` and a `Bag` of EdgeLists, where `EdgeList` is an implicitly defined list of neighbors. Finally, Figure 3.4c uses a similar graph declaration, with the simple tweak of creating an array of `N shard` partitions; a `shard` imposes additional constraints on the vertices that are in the shard: vertices are partitioned into intervals such that all vertices with neighbors in a given vertex interval are all available in the same

```
void Vertex::update() {
  foreach(neighbor n)
    distance = f(n.distance);
}

template <typename V>
class Graph {
  Bag<V> vertices;


public:
  void process();
};

int main() {
   Graph<Vertex> g;
   g.read("/path/to/graph");
   g.preprocess(); //Partition
   g.process();
}
```

(a) Graph for Vertex in Figure 3.1b.

```
void Vertex::update() {
  foreach(neighbor n)
    distance = f(n.distance);
}

template <typnam V, typnam E>
class Graph {
  Box<V> vertices;
  Bag<E> edgeLists;

public:
  void process();
};

int main() {
   Graph<Vertex, EdgeList> g;
   g.read("/path/to/graph");
   g.preprocess(); //Partition
   g.process();
}
```

(b) Decoupling Vertices from Edgelists.

```
void Vertex::update() {
  foreach(neighbor n)
    distance = f(n.distance);
}

template <typename V, typename E>
class Graph {
  Box<V> vertexShard[N];
  Bag<E> edgeShard[N];

public:
  void processShard(int);
};

int main() {
   Graph<Vertex, EdgeList> g;
   g.read("/path/to/graph");
   g.createShards(N); //Preprocess
   for(int i=0; i<N; i++)
      g.processShard(i);
}
```

(c) Using Shard representation of graphs.

Figure 3.4: Variations of graph programming, showcasing the ease and versatility of programming with *InfiniMem*, using its high-level API.

shard [75], enabling fewer random accesses by having all vertices' neighbors available before processing each shard. Note that representing shards in *InfiniMem* is very simple.

```
// SZ = SIZEOF_INPUT;
// BSZ = BATCH_SIZE;
// vb = vertices;


Graph<V>::process() {
  for(i=0; i<SZ; i+=BSZ) {
    // fetch a batch
    vb=fetchBatch(ID(i), BSZ);




    for(j=0; j<BSZ; j++)
      vb[j].update();

    storeBatch(vb, BSZ);
  }
}
```

(a) `process()`-ing graph in Figure 3.1b.

```
// SZ = SIZEOF_INPUT;
// BSZ = BATCH_SIZE;
// v = vertices;
// e = edgeLists;

Graph<V, E>::process() {
  for(i=0; i<SZ; i+=BSZ) {
    // fetch a batch
    vb=v.fetchBatch(ID(i), BSZ);

    // fetch corr. edgelist
    eb=e.fetchBatch(ID(i), BSZ);

    for(j=0; j<BSZ; j++)
      vb[j].update(eb[j]);

    storeBatch(vb, BSZ);
  }
}
```

(b) `process()` for decoupled Vertex.

```
// NS = NUM_SHARDS;
// SS = SIZEOF_SHARD;
// vs = vertexShard;

Graph<V, E>::process() {
  for(i=0; i<NS; i++) {
    // fetch entire memory shard
    mshrd = vs[i].fetchBatch(.., SS);

    // fetch sliding shards
    for(j=0; j<NS; j++)
      sshrd += vs[j].fetchBatch(..,..);

    sg = buildSubGraph(mshrd, sshrd);

    foreach(v in sg)
      v.update();

    storeBatch(mshrd, SS);
  }
}
```

(c) Custom `process()`-ing for shards.

Figure 3.5: Default and custom overrides for `process()` via low-level *InfiniMem* API.

Figure 3.5a illustrates the default `process()`: objects in the `Box` or `Bag` are read in batches and processed one at a time. For graphs with vertices decoupled from edgelists, vertices and edgelists are read in batches and processed one vertex at a time (Figure 3.5b); batches are concurrently processed. Figure 3.5c illustrates custom shard processing: each memory shard and corresponding sliding shards build the subgraph in memory; then each vertex in the subgraph is processed [75].

3.3 *InfiniMem*'s I/O Efficient Object Representation

We now discuss the I/O efficient representation provided by *InfiniMem*. Specifically, we propose an *Implicitly Indexed* representation for fixed-sized data (`Box`); and an *Explicitly Indexed* representation for variable-sized data (`Bag`).

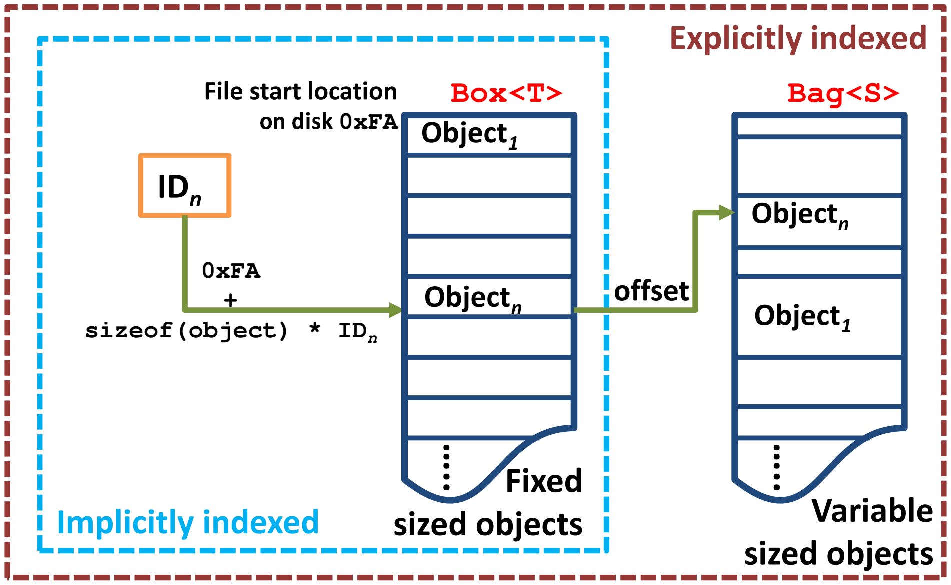


Figure 3.6: Indexed disk representation of fixed- and variable-sized objects.

As the number of objects grows beyond what can be accommodated in main memory, the frequency of object I/O to/from disk storage will increase. This warrants an organization of the disk storage that reduces I/O latency. To allow an object to be addressed regardless of where it resides, it is assigned a unique numeric `ID` from a stream of non-negative, monotonically increasing integers. Figure 3.6 shows the access mechanism for objects using their IDs: fixed-sized data is stored at a location determined by its `ID` and its fixed size: `FILE_START + (sizeof(Object)*ID)`. For variable-sized data, we use a *metafile* whose fixed-sized address entries store the offset of the variable-sized data into the *datafile*. The `Vertex` declared in Figure 3.4a for example, would only use the *explicitly indexed* `Bag` notation to store data, while the representations in Figure 3.4b and Figure 3.4c use both the `Box` and `Bag` for the fixed size `Vertex` and the variable sized `EdgeList` respectively. Thus, fixed-sized data can be fetched/stored in a *single logical* disk seek and variable-sized data in *two logical* seeks. This ensures fetch and store times are nearly constant with *InfiniMem* and independent of the number of objects in the file (like random memory access), and enabling:

– **Efficient access for Fixed-Sized objects:** Using the object `ID` to index into the datafile, *InfiniMem* gives fast access to fixed-sized objects in 1 logical seek.

– **Efficient access for Variable-Sized objects:** The metafile enables fast, random-access access to objects in the datafile, in at most 2 logical seeks.

– **Random Access Disk I/O:** The indexing mechanism provides an imitation of random access to both fixed and variable sized objects on disk.

– **Sequential/Batch Disk I/O:** To read `n` consecutive objects, we seek to the start of the first object. We then read `sizeof(obj)*n` bytes and up to the end of the last object in the sequence for fixed- and variable-sized objects, respectively.

– **Concurrent I/O:** For parallel processing, different objects in the datafile must be concurrently and safely accessed. Given the large number of objects, individual locks for each object would be impractical. Instead, *InfiniMem* provides locks for groups of objects: to decrease lock conflicts, we group non-contiguous objects using modulo `ID` modulo a `MAX_CONCURRENCY` parameter set at 25.

3.4 Evaluation

We now evaluate the programmability and performance of *InfiniMem*. This evaluation is based upon three class of applications: probabilistic web analytics, graph/mesh generation, and graph processing. We also study the scalability of size oblivious applications written using *InfiniMem* with degree of parallelism and input sizes. We programmed size oblivious versions of several applications using *InfiniMem* and are listed in Table 3.1. We begin with data analytics benchmarks: frequency counting using *arrays*, membership query using *hash tables*, and probabilistic membership query using *Bloom filters*. Then, in addition to mesh generation, in this evaluation, we use a variety of graph processing algorithms from diverse domains like graph mining, machine learning, etc. The *Connected Components* (CC) algorithm finds nodes in a graph that are connected to each other by at least one path, with applications in graph theory. *Graph Coloring* (GC) assigns a color to a vertex such that it is distinct from those of all its neighboring vertices with applications in register allocation etc. In a web graph, *PageRank* (PR) [95] iteratively ranks a page based on the ranks of pages with inbound links to the page and is used to rank web search results. *NumPaths* (NP) counts the number of paths between a source and other vertices. From a source node in a graph, *Single Source Shortest Path* (SSSP) finds the shortest path to all other nodes in the graph with applications in logistics and transportation.

3.4.1 Programmability

Application	Additional LoC
Probabilistic Web Analytics	
Frequency Counting	$2 + 3 + 3 = 8$
Membership Query	$2 + 3 + 3 = 8$
Bloom Filter	$2 + 4 + 3 = 9$
Graph/Mesh Generation	
Mesh Generation	$2 + 2 + 2 = 6$
Graph Processing	
Graph Coloring PageRank SSSP Num Paths Connected Components	$1 + 3 + 2 = 6$

Table 3.1: Between 6 and 9 *additional* lines of code are needed to make these applications *size oblivious*. Graph processing uses decoupled version (Figure 3.4b).

Writing size oblivious programs with *InfiniMem* is simple. The programmer needs to only: (a) initialize the *InfiniMem* library, (b) identify the large collections and `Box` or `Bag` them as necessary, and (c) use the default `process()`-ing engine or provide a custom engine. Table 3.1 quantifies the ease of programming with *InfiniMem* by listing the number of additional lines of code for these tasks to make the program size oblivious using the default processing engine. At most 9 lines of code are needed in this case and *InfiniMem* does all the heavy lifting with about 700 lines for the I/O subsystem, and about 900 lines for the runtime, all of which hides the complexity of making data structures disk-resident from the user. Even programming the shard processing framework was relatively easy: about 100 lines for simplistic shard generation and another 200 lines for rest of the processing including loading memory and corresponding sliding shards, building the subgraph in memory and processing the subgraph; rest of the complexity of I/O etc., are handled by *InfiniMem*.

3.4.2 Performance

We now present the runtime performance of applications written using *InfiniMem*. We evaluated *InfiniMem* on a Dell Inspiron machine with 8 cores and 8GB RAM with a commodity 500GB, 7200RPM SATA 3.0 Hitachi HUA722050CLA330 hard drive. For consistency, the disk cache is fully flushed before each run.

Size Oblivious **Graph Processing**

We begin with the evaluation of graph processing applications using input graph datasets with varying number of vertices and edges, listed in Table 3.2. Orkut, Pokec, and LiveJournal graphs are directed graphs representing friend relationships. Vertices in the Amazon graph represent products, while edges represent purchases. The largest input in this evaluation is rmat-805-134 at 14GB on disk, 805M edges and 134M vertices.

Input Graph	$\|V\|$	$\|E\|$	Size
Pokec (PO)	1,632,804	61,245,128	497M
Live Journal (JL)	4,847,571	68,993,773	1.2G
Orkut-2007 (OR)	3,072,627	223,534,301	3.2G
Delicious-UI (DL)	33,778,221	151,772,443	4.2G
RMAT-536-67 (R5)	67,108,864	536,870,912	8.8G
RMAT-805-134 (R8)	134,217,728	805,306,368	14G

Table 3.2: Inputs used in this evaluation.

We first discuss the benefits of decoupling edges from vertices. When vertex data and edgelists are in the same data structure, line 22 in Algorithm 3 requires fetching the edgelists for the vertices even though they are not used in this phase of the computation.

Decoupling the edgelists from vertex data has the benefit of avoiding wasteful I/O: the running times in seconds for these two configurations are shown in Table 3.3. The very large decrease in running time is due to the extremely wasteful I/O that reads the variable sized edgelists along with the vertex data even though only the vertex data is needed.

I/P	PageRank		Conn Comp		Numpaths		Graph Coloring		SSSP	
	Co	DeCo	Co	DeCo	Co	DeCo	Co	DeCo	Co	DeCo
PO	2,228	172	352	60	37	8	277	28	48	7
LJ	8,975	409	1,316	122	106	14	602	58	133	70
OK	3,323	81	3,750	277	459	11	3,046	140	660	154
DL	32,743	1,484	15,404	904	1,112	67	9,524	365	1,453	65
R5	23,588	3,233	12,118	2,545	1,499	861	5,783	1,167	1,853	584
R8	25,698	3,391	>8h	3,380	3,069	1,482	11,332	2,071	>8h	2,882

Table 3.3: Decoupling vertex and edgelists improves performance by avoiding wasteful I/O (time in seconds). 'Co' and 'DeCo' refer to coupled and decoupled respectively.

Figure 3.7 shows the I/O breakdowns for various benchmarks on the moderately sized Delicious-UI input. While the programming effort with *InfiniMem* is already minimal, switching between representations for the same program can be easier too: with as little as *a single change* to data structure definition (figures 3.4a-3.4b), the programmer can evaluate different representations.

Tables 3.4 and 3.5 show the frequencies and percentage of total execution time spent in various I/O operations for processing the *decoupled* graph representation with *InfiniMem*, as illustrated in Figure 3.4b. Observe that the number of batched vertex reads and writes is the same in Table 3.4 since both vertices and edgelists are read together in batches. There are no individual vertex writes since *InfiniMem* only writes vertices in batches. Moreover, the number of batched vertex writes is less than the reads since we write only updated vertices and as the algorithm converges, in some batches, there are no updates.

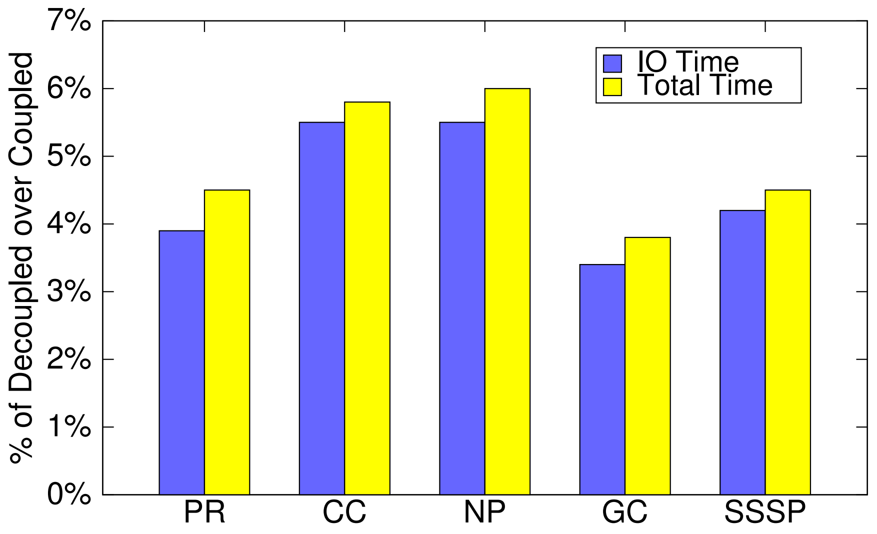


Figure 3.7: Percentage(%) of IO and execution time for decoupled over coupled representations for various applications on the 'Delicious-UI' input.

Observe in Table 3.5 that as described earlier and as expected, the maximum time is spent in random vertex reads.

I/O Operation	LJ	OK	DL	R5	R8
Vertex Batched Reads	7,891	421	40,578	12,481	24,052
Edge Batched Reads	7,891	421	40,578	12,481	24,052
Vertex Individual Reads	865e+6	188e+6	2.8e+9	1.8e+9	2.5e+9
Vertex Batched Writes	7,883	413	40,570	12,473	24,044

Table 3.4: Frequencies of operations for various inputs for PageRank.

I/O Operation	LJ	OK	DL	R5	R8
Vertex Batched Reads	0.05%	0.02%	0.31%	0.12%	0.13%
Edge Batched Reads	8.48%	2.75%	11.25%	7.75%	9.72%
Vertex Individual Reads	54.80%	71.59%	76.96%	86.47%	81.73%
Vertex Batched Writes	0.12%	0.03%	0.37%	0.04%	0.10%
Total IO	63.45%	74.39%	88.89%	94.38%	91.68%

Table 3.5: Percentage of time for I/O operations for various inputs for PageRank.

Sharding with ***InfiniMem***

In the rest of this discussion, we always use the decoupled versions of Vertex and EdgeLists. We now compare various versions of graph processing using *InfiniMem*. Table 3.6 compares the performance of the two simple graph processing frameworks we built on top of *InfiniMem* with that of GraphChi-provided implementations in their 8 thread configuration. *InfiniShard* refers to the shard processing framework based on *InfiniMem*. In general, the slowdown observed with *InfiniMem* is due to the large number of random reads generated, which is O($|E|$). For PageRank with Orkut, however, we see speedup for the following reason: as the iterations progress, the set of changed vertices becomes considerably small: ∼50. So, the number of random reads generated also goes down considerably, speeding up PageRank on the Orkut input. With Connected Components, our *InfiniMem* runs slower primarily because the GraphChi converges in less than half as many iterations on most inputs. Table 3.6 also presents the data for PageRank that processes shards with our *InfiniMem* library as compared to the very fine-tuned GraphChi library. The speedup observed in Table 3.6 from *InfiniMem* to *InfiniShard* is from eliminating random reads enabled by the shard format. Notice that even with our quick, unoptimized ∼350 line implementation of sharding, the average slowdown we see is only 18.7% for PageRank and 22.7% for Connected Components compared to the highly tuned and hand-optimized GraphChi implementation. Therefore, we have shown that *InfiniMem* can be used to easily and quickly provide a *size oblivious programming* experience along with I/O efficiency for quickly evaluating various representations of the same data.

I/P	PageRank Time (sec)			Conn. Comp. Time (sec)		
	InfiniMem (speedup)	*InfiniShard* (speedup)	GraphChi	*InfiniMem* (speedup)	*InfiniShard* (speedup)	GraphChi
PO	172 (0.72)	121 (1.02)	124	60 (0.40)	26 (0.92)	24
LJ	409 (0.90)	488 (0.76)	371	122 (0.49)	80 (0.75)	60
OK	81 (1.91)	190 (0.82)	156	277 (0.44)	142 (0.87)	123
DL	1,484 (0.43)	730 (0.89)	652	904 (0.17)	191 (0.78)	149
R5	3,233 (0.36)	1,637 (0.70)	1,146	2,545 (0.21)	746 (0.71)	529
R8	3,391 (0.44)	2,162 (0.69)	1,492	3,380 (0.30)	1,662 (0.61)	1,016

Table 3.6: *InfiniMem*-Decoupled vs. *InfiniMem*-Sharding vs. GraphChi. The speedups presented are over GraphChi.

Size-Oblivious Programming **of Probabilistic Apps**

Here, we present the throughput numbers for the probabilistic applications in Table 3.7. We evaluated these applications by generating uniformly random numeric input. Frequency counting is evaluated by counting frequencies of random inserts while membership query and Bloom filter are evaluated using uniformly generated random queries on the previously generated uniformly random input. Jenkins hashes are used in Bloom filter. Bloom filter achieves about half the throughput of Frequency Counting since Bloom filter generates twice as many writes.

Application	Throughput (queries/sec)
Frequency Counting	635,031
Membership Query	446,536
Bloom Filter	369,726

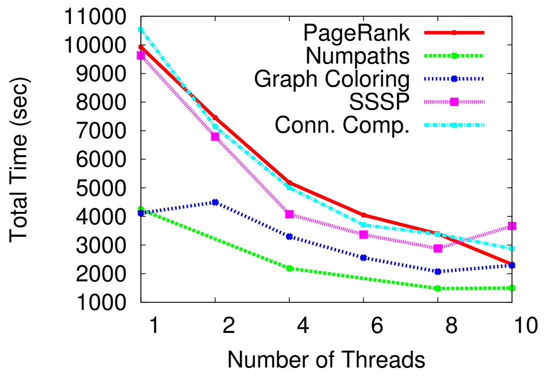
Table 3.7: Throughput for the probabilistic applications.

We also experimented with search/grep. We searched for entries using the Orkut input file (3.2GB on disk) as an input file. Using a naive, sequential scan and search took 67 seconds. Using *InfiniMem* with 1 thread took 15 seconds, while using 4 threads took

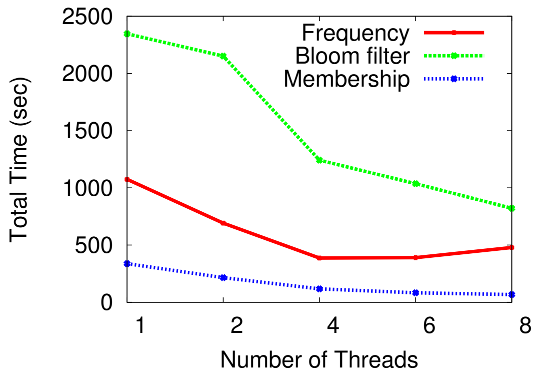
5 seconds for the *same* naive search implementation. The highly optimized GNU Regular Expressions utility took an average of 4.5 seconds for the same search. This shows that in addition to ease of programming, *InfiniMem* delivers performance even with very simple implementations.

3.4.3 Scalability

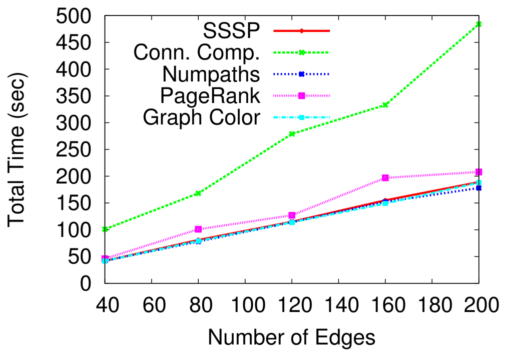
Next, we present data to show that *InfiniMem* scales with increasing parallelism. Figure 3.8a shows the total running times for various applications on the 14GB rmat-805-134 input: for most applications *InfiniMem* scales well up to 8 threads.



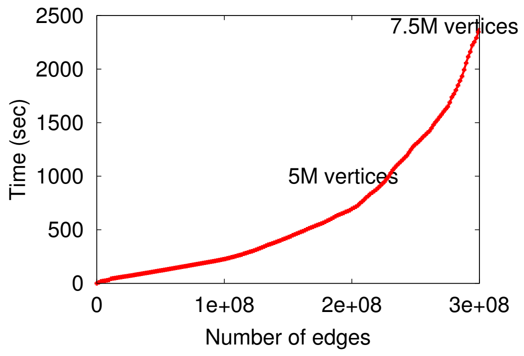
(a) Scalability with parallelism for RMAT-805-134 (14GB)



(b) Scalability with parallelism for Probabilistic Applications



(c) Scalability with $|E|$, 8 threads for various applications.



(d) Generation of a Mesh with 7.5M vertices and 300M edges.

Figure 3.8: Scalability of *InfiniMem* with parallelism and input size.

However, given that the performance of applications is determined by the data representation and the number of random accesses that result in disk I/O, we want to study how well *InfiniMem* scales with increasing input size. To objectively study the scalability with increasing number of edges with fixed vertices and controlling for variations in distribution of vertex degrees and other input graph characteristics, we perform a controlled experiment where we resort to synthetic inputs with 4M vertices and 40M, 80M, 120M, 160M and 200M edges. Figure 3.8c shows the time for each of the for these inputs. We see that with increasing parallelism, *InfiniMem* scales well for increasing number of edges in the graph. This shows that *InfiniMem* effectively manages the limited memory resource by orchestrating seamless offloading to disk as required by the application. The performance on real-world graphs is determined by specific characteristics of the graph like distribution of degrees of the vertices etc. But for a graph of a specified size, Figure 3.8c can be viewed as a practical upper bound.

Figure 3.8b illustrates the scalability achievable with programming with *InfiniMem* with parallelism for the Frequency counting, Exact membership query and Probabilistic membership query using Bloom filters. Notice that these applications scale well with increasing number of threads as well as increasing input sizes. The execution time for Bloom filter is significantly larger since Bloom filter generates more random writes, depending on the number of hash functions utilized by the filter; here, two independent hashes are used.

Figure 3.8d illustrates that very large graph generation is feasible with *InfiniMem* by showing the generation of a Mesh with 7.5M vertices and 300M edges which takes about 40 minutes (2400 seconds). We observe that up to 5M vertices and 200M edges, the time for generation increases nearly linearly with the number of edges generated after which the generation begins to slow down. This slowdown is not due to the inherent complexity of generating larger graphs: the number of type of disk operations needed to add edges is independent of the size of the graph – edge addition entails adding the vertex as the neighbor's neighbor and accessing the desired data in *InfiniMem* requires a maximum of 2 logical seeks. Instead, the slowdown is because modifications of variable sized data structures in *InfiniMem* are appended to the datafile on disk, which grows very large over time and the disk caching mechanisms begin to get less effective. Compare this with the fact that GTGraph crashed immediately for a graph with just 1M vertices and 400M edges.

3.5 Summary

This chapter presented the *InfiniMem* system for enabling *size oblivious programming.* The techniques developed in this chapter are incorporated in the versatile general purpose *InfiniMem* library. In addition to various general purpose programs, we also built two more very domain specific graph processing frameworks on top of *InfiniMem* including processing GraphChi-style shards. *InfiniMem* performance scales well with parallelism, increasing input size highlighting the necessity of concurrent I/O design in a parallel set up. Our experiments show that *InfiniMem* can successfully generate a graph with 7.5M vertices and 300M edges (4.5 GB on disk) in 40 minutes and compute PageRank on an RMAT graph with 134M vertices and 805M edges (14GB on disk) on an 8-core machine in ∼54 minutes.

Chapter 4

ABC^2: Adaptively Balancing Computation and Communication in DSM Clusters

Big-data applications have become increasingly important in many application domains. The large inputs offer data level parallelism at a scale that makes it attractive to run such applications on distributed shared memory (DSM) based modern clusters composed of multicore machines. Clusters offer an attractive computing platform for achieving scalable performance on data and compute intensive applications. To simplify the task of programming clusters, distributed shared memory (DSM) has been widely used. The memory resources available across the machines in a cluster are harnessed as one and made available to the application in form of shared-memory. The large amount of application data stored in DSM is actually scattered across the machines and must be transferred across them as needed by the computation. A variety of DSMs have been built in the past [8, 58, 112]. These DSMs were developed before the advent of multicore machines. To deliver high performance, latency hiding mechanisms were deployed to mitigate the impact of communication delays associated with transferring data across machines: for example, creating multiple application threads was the common approach. The primary limitation of this approach is that computation and communication are still coupled: the communication is still in the *critical path of computation*. Moreover, with the advent of multi-cores, a much larger number of application threads becomes feasible – but that also increases communication which can quickly become the bottleneck on DSMs.

Our analysis of several applications that require software speculation for correct parallel execution shows that the balance between computation and communication differs between applications. Decoupling communication and computation will allow for dynamically balancing computation and communication – this will automatically achieve optimal performance even in the face of changes to the DSM/cluster HW/SW configuration. In this work, we study this balance in the context of DSM for applications that benefit from speculative parallelism [40, 94, 106].

In this chapter, we study this balance in the context of DSMs and exploit the multiple cores present in modern multicore machines by creating three kinds of threads which allows us to dynamically balance computation and communication: compute threads to exploit data level parallelism in the computation; fetch threads that replicate data into object-stores before it is accessed by compute threads; and update threads that make results computed by compute threads visible to all compute threads by writing them to DSM. We observe that the best configuration for above mechanisms varies across different inputs in addition to the variation across different applications. To this end, we design ABC^2: a runtime algorithm that automatically configures the DSM using simple runtime information such as: observed object prefetch and update queue lengths. This runtime algorithm achieves speedups close to that of the best hand-optimized configurations.

4.1 The Need for Distributed Software Speculation

Speculative parallelism is a software technique that parallelizes algorithms by creating parallel threads to perform computations that may exhibit non-deterministic sharing of data [40, 94, 106]. Irregular applications, such as vertex centric graph processing algorithms, contain significant levels of data parallelism as the same operation needs to be performed repeatedly over the large number of vertices in the graph. On a distributed system, the vertices and their processing can be distributed across the machines to exploit data parallelism. While use of DSM makes the task of programming easy and the distributed caching protocol reduces long latency fetch operations, the irregular nature of parallelism makes it necessary to employ speculation. Infrequently arising data dependences requires processing of interdependent vertices to be serialized. While these dependences are infrequent, they are dependent upon the input data set, and hence are revealed at runtime. Speculation involves processing vertices in parallel assuming that there are no dependences between them, runtime detection of misspeculation when the dependences arise, and then recovering from them. As long as misspeculation rate is low, the benefits of speculation far outweigh the cost of recovering from misspeculation.

Techniques for exploiting data-parallelism have been extensively researched, but parallelizing *irregular applications* is still work in progress [103, 70]. *Irregular applications* or applications with *irregular parallelism* are typically those that process structures like graphs and cannot be statically analyzed for parallelism the way regular data-parallel applications can [70]. The parallelism present in these applications is actually input dependent, which can vary from one run to the next. For example, a road network is a sparse graph while a social network is often a power law graph. Parallelizing applications that handle diverse types of inputs requires dynamic techniques like speculation.

Software *speculation* [105, 104] is a runtime approach to loop parallelization which *speculates* that there are no cross iteration data dependencies and treats the loop as if it were a *doall* loop. Once the computation is complete, a *misspeculation detection* phase checks if any data dependency violations have actually occurred. If not, the computation can commit the computed value; in case of a violation, the result of the computation is discarded and the computation restarted. Speculation is accomplished by privatization of data [105]: a copy of the data is saved in a compute-private buffer to isolate the computation from other concurrently executing computations. The modified data in the thread-private buffer is then speculatively committed back to memory. This approach has also been applied recently in SpiceC [40] for shared as well as distributed memory software speculation [30].

As an example consider the pseudocode for *graph coloring* listed in Algorithm 4.1. To parallelize the loop on Line 2 requires speculation since the list of vertex's neighbors can only be known at runtime. Speculation involves treating this for loop as a *doall* loop and executing it in parallel. When the vertices being processed in parallel are not neighbors, no misspeculation is encountered. However, if neighbors are processed in parallel, they may be assigned the same color leading to misspeculation. A commit phase that updates the colors of vertices detects this misspeculation and initiates recovery. The recovery requires repeating the coloring process for a vertex. If misspeculation occurs infrequently, speculative parallelization results in speedups.

Algorithm 4: Vertex-centric sequential graph coloring.

```
1 Graph g;
2 for int i=0; i<g.size(); i++ do
3     Vertex v = g[i];
4     v.color = 0;
5     for int j=0; j<v.numNeighbors(); j++ do
6         if v.nbr[j].color == v.color then
7             v.color++;
```

As shown in Figure 4.1, the node N_i being colored by thread T_i is the neighbor of the node N_j which is concurrently being colored by thread T_j. Therefore, the coloring information of node N_i with thread T_j and node N_j with thread T_i may indicate that the two nodes are not colored. This can lead to incorrect coloring assignment, i.e. the two nodes may be assigned the same color.

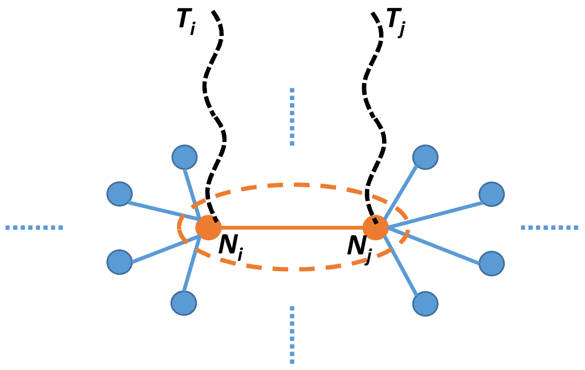


Figure 4.1: Dependencies between neighboring nodes requires speculation in order to correctly parallelize graph coloring.

On the face of it, it appears that an algorithm such as graph coloring cannot be parallelized. However, this is where speculation comes in. With speculation, there are two distinct phases: (1) computation and (2) commit. To begin, each computation thread *speculates* that the data it reads will stay current during the computation and proceeds with the computation and stores the results privately. Once the computation is complete, the thread attempts to commit the results in order to make these results visible to other computation threads. As part of the commit phase, a *mis-speculation check* is performed to assert that the data read by the thread is, in fact, still current. The result of a speculative computation is committed *only* if the mis-speculation check succeeds. On failure, the result of the computation is discarded and the computation is scheduled to be re-executed.

Therefore, in the presence of potential sharing between concurrently executing threads, the key idea behind speculation is to *speculate that such sharing does not occur* and proceed with the computation. Once the results of the computation are ready to be written to memory, a mis-speculation check is performed to assert the absence of such sharing. This can be achieved by resorting to versioning of data, for example. Thus, speculative parallelism only commits those results that are computed from most current values. With speculative parallelism, data commit failures can arise from violating write-read or read-write or write-write dependencies that manifest at runtime. In this work, we use the SpiceC [40] speculative execution model.

Software speculation has been been successfully applied to parallelize irregular applications for distributed shared memory systems [105, 40]. However, distributed software speculation has not been fully researched. In this work we study the interaction between caching and speculation. In the presence of a cache, every read from the local cache in the DSM can cause a misspeculation because, due to network latency, even though a cached object may have been updated on the remote machine, the invalidate may not have been received at the local cache before the object is speculatively read. Therefore, in a DSM with caching, larger caches can lead to more misspeculations.

For speculation to be effective, the sharing of nodes between concurrently executing threads should be small. If such sharing is high, the potential benefits from speculation will be offset by repeated re-computations caused by commit failures that result from failed mis-speculation checks. Examples of applications that have low sharing and can benefit from speculative execution include: Graph Coloring (GC), Single Source Shortest Path (SSSP), KMeans (KM) and Maximal Independent Set (MIS).

4.2 Communication Bottleneck on Multicore Machines

While clusters have an inherently scalable architecture, shared memory systems are easier to program than clusters since there is only one memory and address space to manage. To make programming of clusters easy, the Distributed Shared Memory (DSM) paradigm has been widely used. With DSMs, the programmer uses the familiar shared memory paradigm, while the underlying DSM runtime transparently translates and handles remote load/store requests and via caching it minimizes the need for long latency fetch operations. DSM systems like the object based Orca [8] and linear-memory based Shasta [112] leverage *caching* and *prefetching*, via automated and programmable prefetch-hints, to tolerate communication latencies.

Since each machine in a modern cluster supports multiple cores, simultaneous requests for communication originating from multiple computation threads can rapidly cause the communication to become a performance bottleneck – if programs designed for prior DSMs and developed for a cluster of uniprocessor machines are naively executed on modern multicore machines by simply running more threads on each machine, the network becomes the bottleneck. For example, on a cluster of six 8-core machines, we observed that a 5x increase in the number of threads generating communication requests resulted in a 7x increase in fetch time from the DSM. In this work we exploit multicore machines to tolerate communication latency by introducing dedicated communication threads and move the communication latency off the critical path.

Applications generate high-priority *fetch requests* when the node to be processed is needed – these must be serviced immediately and the requesting thread blocks until this request completes. Once a node has been fetched, additional requests are issued to *prefetch* the neighbors. Finally, once the speculative computation is completed, an *update request* is issued to commit the updated private copy of the data. The update thread performs the mis-speculation check and performs the commit asynchronously and off the critical path of the speculative computation.

Computation being performed generates two types of network requests: *fetch* and *update*. In addition, to tolerate fetch latency, *prefetch* requests may also be issued. Fetch requests are of the highest priority since a computation must stall on a fetch. On the other hand, *prefetch* requests can be given a lower priority because they are issued to reduce latency of future fetch requests and *update* requests can be given lower priority because there may or may not be other computations waiting on the updates. Therefore, depending upon the spare network capacity, *prefetch* requests and *update* requests should be accommodated at a rate that does not overwhelm the network. Moreover, depending upon the needs of the application, a balance between handling of prefetch and update requests should be maintained. In this work, for each machine, the number of outstanding *prefetch* requests and outstanding *update* requests is controlled to: limit the amount of network capacity they use; and to maintain a balance between the two types of requests. We develop a system to achieve this goal that, for a given number of *compute* threads, dynamically varies the number of *prefetch* threads and *update* threads to meet the needs of the application.

As an example, our experiments show that merely increasing the number of compute threads can actually hurt the performance of distributed Graph Coloring – on a cluster of 6 machines, the speedup with 4 computation threads and 4 prefetch threads was 2.4x that of the speedup with 8 computation threads and 4 prefetch threads. We further observed that blindly increasing either prefetch or update threads in DSM based programs also can hurt performance. An optimal configuration that uses just 4 compute threads per machine and dynamically varies the number of update threads was faster than the case of 4 compute threads with 4 prefetchers by a factor of 1.2x. To summarize, this data illustrates that fewer computation threads when balanced with communication gives better speedup compared to more computation threads. Clearly, this data is an indicator for the need to carefully balance computation and communication in DSM based applications.

4.3 Dynamically Adaptive Communication

This latency tolerance mechanism hides communication latency by creating communication threads that run concurrently with computation on the multicore machine. Two types of communication threads are used: *prefetch threads* and *update threads.* Prefetch threads fetch objects from the DSM into the prefetch buffer before they are accessed by the threads on a machine so that computation threads can avoid potential long access latency. Update threads are responsible for writing modified objects to the DSM so that computation threads can proceed with execution without waiting for object updates in DSM to complete. Given a fixed number of computation threads, the number of communication threads required will depend upon the rate at which the computation threads initiate fetch and update operations from and to the DSM. Therefore we must decide:

- *Degree of prefetching* - determined by the number of prefetch threads that are created to run concurrently with the given number of computation threads; and

- *Aggressiveness of updates* - determined by the number of update threads that are created to run concurrently with computation threads.

An appropriate balance between computation and communication threads must be dynamically maintained at runtime to optimize performance.

4.3.1 Motivating Study

In this section, we present results of a study that motivates the conjecture that different applications require different balance between communication and computation. Table 5.1 lists the applications considered in this work and their characteristics. For this study, we consider applications with speculative parallelism [40, 94, 106]. We briefly describe these classes of applications in subsection Section 4.3.2. Observe that that applications with low sharing benefit from speculation. This is because when sharing between nodes is low, the runtime dependencies are fewere and therefore the benefits from speculation are higher than its overheads.

Application	Speculative Parallelism	Low Sharing
Graph Coloring (GC)	√	√
KMeans (KM)	√	√
Single Src Shortest Path (SSSP)	√	√
Maximal Independent Set (MIS)	√	√

Table 4.1: A Suite of Modern Irregular Applications

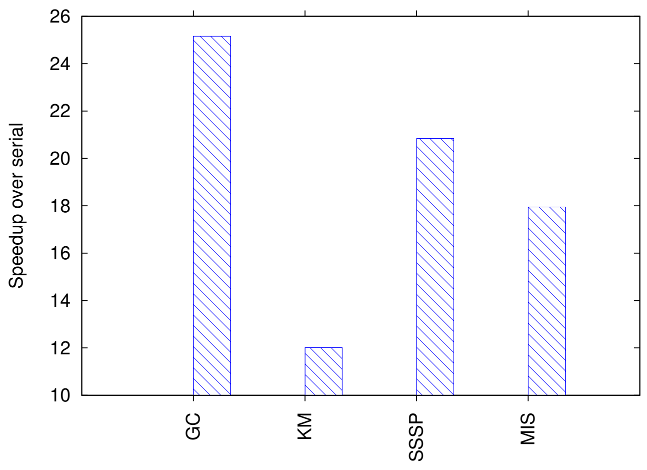


Figure 4.2: Speedups of the parallel versions of the benchmarks over their serial versions.

We first present the raw speedups of the parallel versions (without using our techniques) over the serial versions. The experiments were conducted on an object-based DSM we built system [63], which uses the SpiceC's copy-in copy-out model [40] for speculative execution. Figure 4.2 shows that before using our techniques, the speculative benchmarks achieve 30x speedup on average. Our goal is to *further* improve the speedups of these parallel benchmarks.

In the rest of this study, the speedups we present are over the baseline parallel version that does not use separate prefetching or update threads. This allows us to objectively study the speedups that can be attributed exclusively to the mechanisms considered in this study.

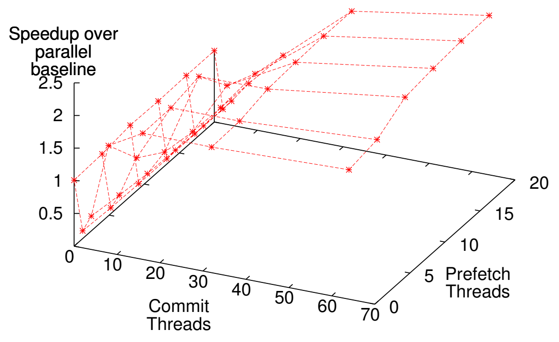
In the remainder of this section we study the benefits of our techniques on speculative parallelism observed in applications listed in Table 5.1 and evaluate the benefits from balancing communication and computation. As mentioned above, the baseline used is the parallel version without our techniques. This allows us to evaluate the specific benefit from using dedicated prefetch and update threads and dynamically adapting these resources.

4.3.2 Adapting Communication for Distributed Speculative Parallelism

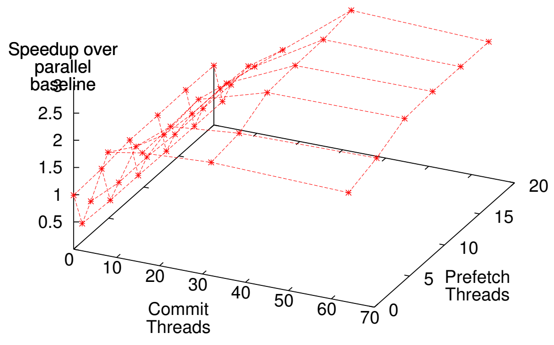
We now present a study of the speculative benchmarks listed in Figure 5.1. Figure 4.3 show the speedups Graph Coloring (GC), Single Source Shortest Path (SSSP), KMeans (KM), and Maximal Independent Set (MIS) with varying number of commit and prefetch threads on the lower axes. The baseline for the speedups is the configuration with zero prefetch and zero update threads. We now present the observations for communication.

Communication Strategy

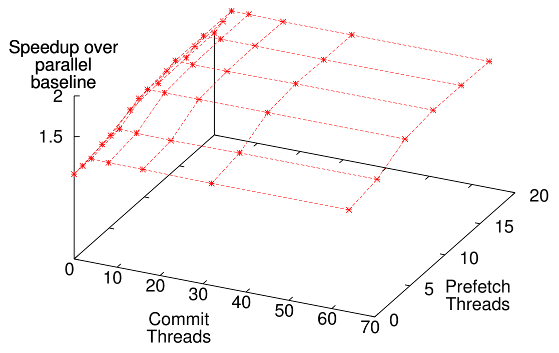
Asynchronously prefetching the data needed for computation in the near future can definitely help speedup the computations: this is seen in the speedups plotted in Figures 4.3a,



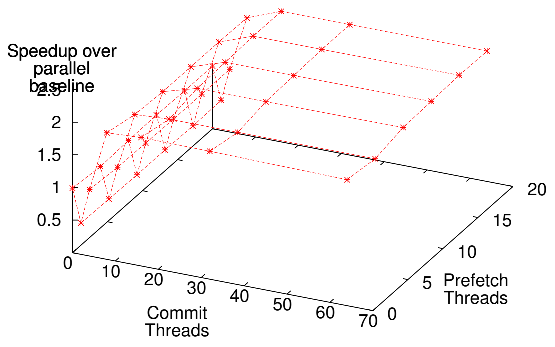
(a) GC: Speedup



(b) SSSP: Speedup

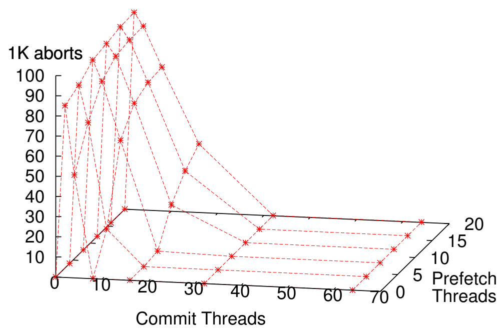


(c) KM: Speedup

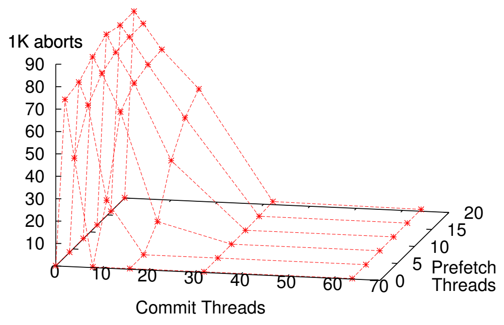


(d) MIS: Speedup

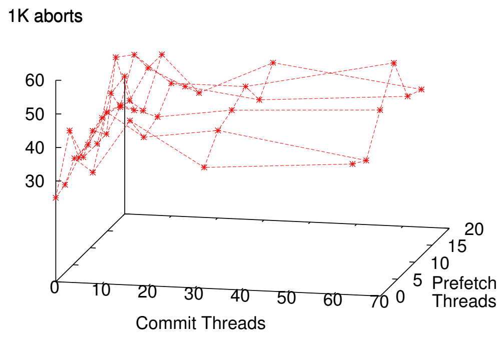
Figure 4.3: Figures 4.3a, 4.3b, 4.3c and 4.3d show the speedups for GC, SSSP, KM and MIS respectively. Prefetch threads make remote objects locally available, hence avoiding remote accesses and reducing the average fetch time. Each benchmark was executed on a cluster of 6 machines, running 4 computation threads.



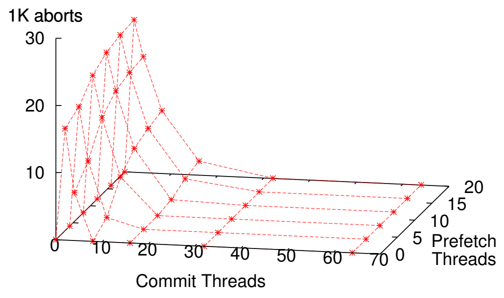
(a) GC: Aborts



(b) SSSP: Aborts



(c) KM: Aborts



(d) MIS: Aborts

Figure 4.4: Figures 4.4a, 4.4b, 4.4c and 4.4d show the number of 1K aborts due to misspeculation for GC, SSSP, KM and MIS respectively. Lesser number of commit threads delay the availability of newly computed object values resulting in higher aborts of computations on stale data values.

4.3b, 4.3c and 4.3d. This is especially true when there are a proportionately large number of neighbors to process for each node. In addition, speculation will benefit from aggressive updates back to the DSM: this is apparent from the increasing speedup with an increasing number of commit threads in Figure 4.3. Delaying commits increases the mis-speculation rate as the computation threads continue to compute results based on potentially stale values. This can be observed in Figure 4.4 – the aborts are higher for fewer commit threads and rapidly decrease with increasing commit threads. Further, notice that in most cases when there are fewer commit threads, aggressive prefetching can result in a 10x increase in aborts. This is because prefetching aggressively brings in stale values since the newer values are still in the update queue. As a result, all computations that speculated on prefetched stale values *will* eventually abort, thereby resulting in an overall slow down of the program execution. Therefore, it is imperative that depending on the speed of computation, the number of update threads – both prefetch and commit – should be dynamically balanced as needed to speedup commits.

Notice that neither the number of prefetch threads nor the number of commit threads can be statically determined in advance: the number of most effective prefetch threads depends on the number of compute threads, the speed of the computation, dynamic network latencies etc. All these point to the need for dynamically adapting these types of requests.

In the next section, we present an adaptive computational model that directly follows from the above observations. We propose, present and evaluate various parameters that can be monitored at runtime. Finally, we evaluate our proposed dynamic model and show that it achieves speedups close to that of the best hand-optimized parallel versions.

4.4 ABC^2: An Adaptive Runtime Framework

The results of the study of applications in the previous section are summarized in Table 4.2. As we can see, not all applications benefit from prefetching but for those that do, the appropriate number of prefetch threads can vary. Finally, although update threads help tolerate latency in all applications, the appropriate number of update threads varies across applications. Therefore we conclude that to benefit from the latency tolerance mechanisms we support, it is best to develop a runtime model that supports all of the proposed mechanisms and, with the help of runtime monitoring, it adapts their use to meet the needs of the application.

Strategy	Speculative Parallelism
Number of Prefetch Threads	Varying
Number of Update Threads	Varying

Table 4.2: Summary of communication strategies.

In this section we develop an adaptive framework that performs well for all different types of parallel applications considered without any a priori knowledge of their type or behavior. We identify parameters that are monitored at runtime to guide and control the degree of prefetching and aggressiveness of updates. In the remainder of this section we present the system architecture and design, discuss the parameters monitored at runtime, and finally present ABC^2 which is the runtime decision making algorithm.

4.4.1 System Design

In this section we present the system design and describe the roles and responsibilities of its various components.

DSM. We built and used an object-based DSM [63] as the underlying DSM in this study.

Speculation. As described in Section 4.3.2, we have implemented the copy-in, copy-out speculative model from SpiceC [40] into the object-based DSM. The separate compute and commit phases of this model are in tune with our need to separate computation and communication.

Thread Pool. To eliminate the penalty of creating and terminating new threads at runtime, we employ a *thread pool* model. A large number of threads of each required type are created during the runtime initialization. Unused threads are put to sleep and woken up as needed rather than resorting to polling for work. This prevents idle threads from using CPU resources.

Prefetching. Prefetching is implemented to take advantage of the graph structures being used by the applications. The computation threads enque the objects IDs of the objects that need to be prefetched into the *prefetch queue*. The prefetch threads first deque the object IDs from the prefetch queue, then fetch the object asynchronously and place the object into a *prefetch buffer*. When per-machine replication is used, the per-machine replica store can be used as the prefetch buffer. When no replication is used, a separate prefetch buffer needs to be used.

Updates. Update threads first dequeue the thread-private memory object from the commit queue and asynchronously *commit* the data back to the DSM. With speculation, the update thread must *atomically* perform the mis-speculation check (discussed in Section 4.3.2) that involves detecting write-read, read-write, and write-write dependencies for all the data in the thread-private memory object and **then** perform the write back to the DSM. Therefore, when speculation is used, the update threads work in a *commit* mode. Note that since the mis-speculation check and update into the DSM need to be performed atomically, batch updates into the DSM can be performed efficiently.

Computation. Computations are performed by compute threads. When speculation is used, the compute threads implement the copy-in, copy-out model to allow concurrent compute threads to execute in isolation. Under this model, the compute threads copy all the data used by the computation into a thread-local private memory. Once the computation is complete, the entire private memory (which also contains the results of the computation) are pushed into a commit queue, to be handled asynchronously by the update threads. Similar mechanism is also used for applications with asynchronous parallelism. Figure 4.5 summarizes the overview of the prototype. The core components of the framework are:

1. Object based DSM;
2. Speculation via SpiceC's copy-in, copy-out model;
3. Separate computation and commit phases; and
4. Prefetch, Compute, and Update thread pools.

The numbering in Figure 4.5 indicates the flow of execution. The compute thread first populates the prefetch queue with the IDs of objects to be prefetched and returns to its computation task. The prefetch threads asynchronously fetch the necessary data from the DSM into the prefetch-buffer. When the compute thread needs a data item, it first looks up the prefetch buffer and brings it into the per-machine replication stores as appropriate; the compute thread then continues with its task. Once completed, the compute thread pushes the results of the computation into the update queue and starts working on the next computation. The update threads dequeue the data from the update queue and commit or discard the results of the computation depending on the success or failure of the mis-speculation check.

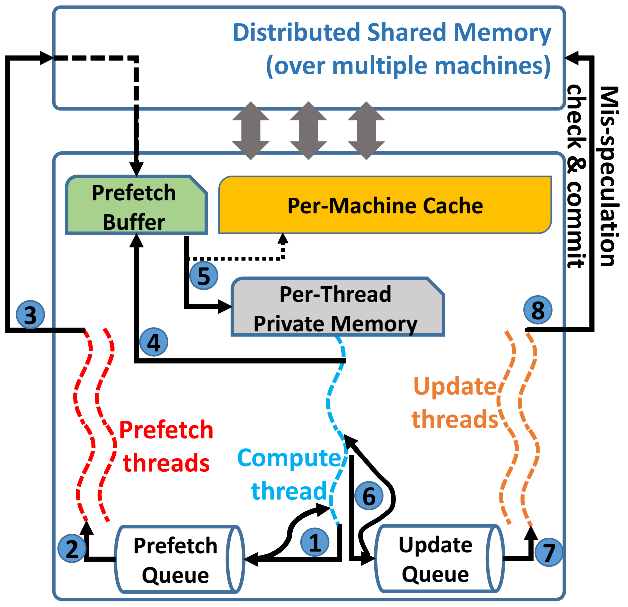


Figure 4.5: The ABC^2 system design showing the DSM, Prefetch, Compute & Update threads.

```
Adapt Prefetching:
// At the start of the prefetch task:
0. if(prefetch_q.length > PQ_Threshold):
1.     wakeup_more_prefetchers()
2. else:
3.     sleep_extra_prefetchers()

Adapt Updates:
// At the start of update task:
0. if(update_q.length > UQ_Threshold):
1.    wakeup_more_udapters()
2. else:
3.    sleep_extra_updaters()
```

Figure 4.6: The ABC^2 Algorithm.

4.4.2 The ABC^2 Algorithm

For each of the decisions the runtime needs to make, we consider the various parameters that we evaluated in Section 4.3.1. To vary the number of prefetch threads, we propose to monitor the length of the prefetch queue, which contains the IDs of objects that need to be prefetched. Finally, to vary the number of update threads, we monitor the length of update queue, which contains the results of computations to be written back to the DSM.

The adaptive ABC^2 algorithm monitors the various parameters listed above to adapt the computation model and communication resources. There are two independent parts to the algorithm, both of which are initiated simultaneously at start of the application. The two independent parts adaptively control the prefetch threads, and update threads. The listing in Figure 4.6 presents the adaptive algorithm. To adapt prefetching and updates, more threads are used depending upon the current queue sizes. In these experiments, the maximum number of prefetch threads was experimentally bounded at 25 and the number of update threads was bounded at 64.

Each of the parameters like prefetch and update queue lengths are monitored as needed in Figure 4.6. We now briefly discuss the benefits and generality of monitoring these parameters.

1. *Prefetch-queue length*: To turn on or turn off prefetching on demand, we simply leverage the presence of the prefetch queue. We will always have at least one prefetcher active. If this prefetcher observes many pending data objects to be prefetched, by querying the prefetch queue length, it will wake up more prefetchers. If prefetchers find no work to do, they simply go to sleep until woken up again.

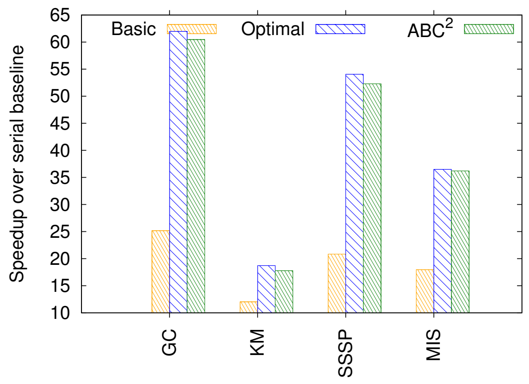
2. *Update-queue length*: Update threads are dynamically adjusted in a manner similar to that used for prefetch threads, with the exception that the update threads monitor the length of the update queue.

4.4.3 Evaluation of ABC^2

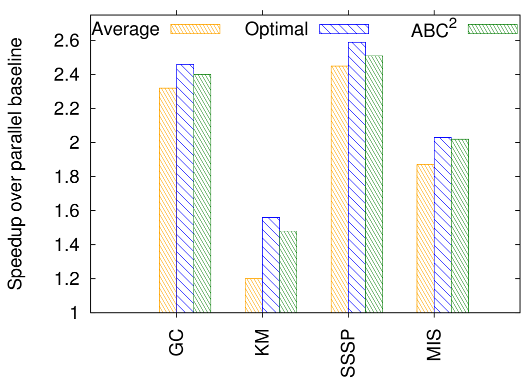
We now evaluate the ABC^2 framework on a cluster of 6 8-core Dell T410 machines each with 8 GB memory, running Ubuntu 10.04, Kernel v2.6.32-21. Our goal is to compare the speedups achieved by the adaptive framework with those of the fastest configurations from the study in Section 4.3.2. The speculative benchmarks are run with 4 compute threads, while adaptively varying prefetch and update threads. In this evaluation, we consider the following different configurations:

- `Basic`, which is the basic configuration *without* prefetch or update threads: without ABC^2, prefetching and updates are handled by the computation thread itself.
- `Optimal`, which is the fastest configuration from the motivation in Section 4.3.2.
- `Average`, where the maximum number of prefetch and update threads is set to the respective average numbers as measured in the ABC^2 version.

We show that for each benchmark, our adaptive framework (a) automatically selects the optimal fastest configuration seen in the study above; and (b) achieves speedups comparable to the fastest configuration.



(a) Speedups without and with ABC^2.



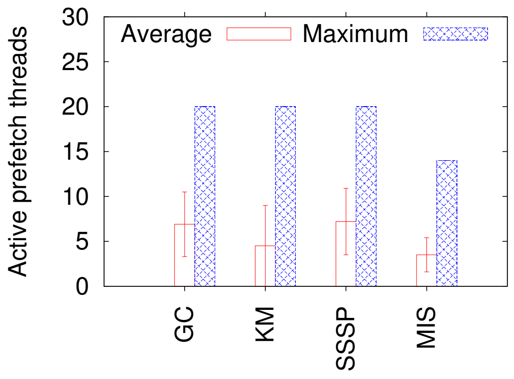
(b) Performance evaluation of ABC^2.

Speedup over serial baseline

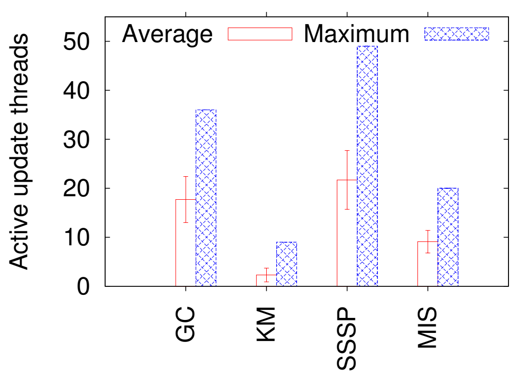
We first show that parallel benchmarks running on 6 machines with tt ABC^2 achieve significant speedups over the `Basic` version without ABC^2. The speedups in this experiment are baselined to the serial versions of their benchmarks. Both the serial and parallel versions fetch data from the DSM and update data back into the DSM. In the serial versions of the benchmarks, the network communication is handled by the computation thread itself. Figure 4.7a compares the speedups of the `Basic`, `Optimal` and ABC^2. First, we see that without ABC^2, the `Basic` versions of speculative benchmarks, on an average, achieve speedups of 18x. Next, we see that the best achievable speedups with prefetch and update threads from the study in Section 4.3.1 is significantly higher: an average of 40x for the speculative benchmarks. On an average, this is an improvement over the `Basic` version by 22x for speculative benchmarks. This clearly shows that ABC^2 algorithm is beneficial for performance. This apparently very large increase is because these benchmarks are mostly network-bound due to the DSM communication. Therefore, providing dedicated resources for network IO in the form of prefetch and update threads hides the network communication latency while simultaneously allowing the computations to make progress with overall speedup.

Speedup over parallel baseline

We now evaluate the speedup benefits of ABC^2 for the *parallel* versions of the benchmarks. The baseline in this experiment is the parallel benchmark running on 6 machines without dedicated prefetching or update threads; the fetches and updates are handled by the computation thread itself. This baseline allows us to evaluate the specific benefit of the ABC^2 algorithm dynamically adapting the prefetch and update threads for parallel programs. Therefore, the speedups presented here are over and above the speedups achieved by the parallel program without adaptive communication. Figure 4.7b compares the speedups of the `Optimal` configuration from Section 4.3.1 with the speedups achieved by the ABC^2 algorithm. We see that the ABC^2 algorithm achieves speedups between 1-6% of that achieved in the study.



(c) Concurrently active *prefetch* threads.



(d) Concurrently active *update* threads.

Adaptive Prefetching and Updates

Finally, we evaluate the specific improvements accrued by ABC^2 adaptively varying prefetch and update threads. Comparing the speedups achieved using the average numbers for prefetch and update threads obtained from Figures 4.7c and 4.7d with the speedups obtained by the ABC^2, we see that ABC^2 always gives better speedups close to the optimal in the study. This indicates that a higher maximum number for prefetch and update threads employed by ABC^2 is important and useful in achieving better speedups.

Figures 4.7c and 4.7d show the average and maximum number of concurrently active prefetch and update threads as measured by the ABC^2 algorithm. We see that the ABC^2 algorithm uses, on an average, 5 prefetch threads. On an average, all the asynchronous benchmarks use just 2 update threads while the speculative benchmarks use 16 update threads, except for KM, which uses only 2 update threads. The update phase of asynchronous benchmarks is significantly shorter compared to the speculative benchmarks: for GC, SSSP and MIS, since a node's value is calculated based on its neighbors, the mis-speculation check involves the node and all its neighbors. Therefore, speculative updates are network-bound and time consuming, but not CPU-bound. Hence, with fewer update threads, more commit requests can get queued to the update queue. But as can be seen from the `Adapt Updates` section in Figure 4.6, when the update queue length is greater than a threshold, the ABC^2 algorithm wakes up more update threads. This does not happen often with asynchronous benchmarks since there is no mis-speculation check in their updates. With KM, the mis-speculation check only involves the single cluster data; there is no notion of neighbors in this case. Therefore, the mis-speculation check is fast and hence fewer number of update threads are sufficient to keep the update queue length below the set threshold. Also, the maximum numbers show that occasionally the ABC^2 algorithm uses a maximum of 25 prefetch threads and 49 update threads. The vertical bars show the standard deviation around the average number of prefetch and update threads. The average, standard deviation and the maximum are indicative of the dynamically varying number of prefetch and update threads in the ABC^2 algorithm. The standard errors for the average number of prefetch and update threads are 0.0036 and 0.0054, indicating a very high confidence in the average numbers shown.

4.5 Summary

In this paper, we motivate the need to delicately balance computation and communication for applications with speculative and asynchronous parallelism. To address this problem, we propose to enable fine-tuned balance between computation and communication: we propose the separation of concerns into prefetch, compute and update threads. To dynamically adapt the system based on the runtime application and data characteristics, we proposed a scheme to monitor data sharing, hit-rates and fetch-times. An evaluation of the ABC^2 model shows that the adaptive scheme achieves performance close to that of the optimal case.

Chapter 5

Distributed Software Speculation on Caching DSMs

Clusters with caching DSMs deliver programmability and performance by supporting shared-memory programming model and tolerating communication latency of remote fetches via caching. The input of a data parallel program is partitioned across machines in the cluster while the DSM transparently fetches and caches remote data as needed by the application. Irregular applications are challenging to parallelize because the input related data dependences that manifest at runtime require the use of *speculation* for correct parallel execution. By speculating that there are no cross iteration dependences, multiple iterations of a data parallel loop are executed in parallel using locally cached copies of data; the absence of dependences is validated before committing the speculatively computed results.

This chapter shows that in *irregular* data-parallel applications, while caching helps tolerate long communication latencies, using a value read from the cache in a computation can lead to misspeculation and thus aggressive caching can degrade performance due to increased misspeculation rate. To limit misspeculation rate we present optimizations for distributed speculation and caching based DSM that decrease the cost of misspeculation check and speed up the re-execution of misspeculated recomputations. These optimizations give speedups of 2.24x for graph coloring, 1.71x for connected components, 1.88x for community detection, 1.32x for shortest path, and 1.74x for pagerank over baseline parallel executions.

5.1 Speculation on Caching-based DSM Systems

While DSMs coupled with caching work well for non-speculative data-parallel applications, aggressive caching can hurt performance during speculative execution due to the potential of using stale values in computations. With distributed speculation on a caching DSM, every cache read can cause misspeculation because the read could occur before a network-delayed invalidate is received – note that successful speculation is counting on the fact that no invalidate would arrive. In this chapter we show that aggressive caching can degrade performance due to increased misspeculation rate. Furthermore, to prevent an increase in misspeculation rate, we develop optimizations for a distributed speculation and caching based DSM to speedup misspeculation check and re-execution of misspeculated recomputations.

We build upon the DSM, cache and commit protocols recently described by Dash et. al. [30]: the input and address space are disjointly partitioned across all the machines and each data object is assigned a home node; every computation thread maintains a thread-private cache for speculation in addition to the per-machine directory-based object caches. Data is accessed first by looking up the thread-private cache, then in the object cache and fetched from the remote home node on a cache miss. The per-machine object cache is managed similar to the high-performance optimized cache-coherence protocol described in [30] that essentially relies on lazy updates and proactive, directory-initiated invalidates. Finally, we use privatization of data to implement speculative parallelism. While [30] focused on integrating caching and prefetching, we show that aggressive caching can slow down execution due to increased misspeculations caused by stale, cached data. In this work, we seek to optimize the underlying caching protocol for speculation to speed up distributed speculation and hence reduce the likelihood of misspeculations.

In the rest of this chapter, we first present the protocol for distributed software speculation on a caching DSM. We then analyze the protocol to *derive optimizations* which we then implement and evaluate. In particular, we identify the following optimizations to the protocol: (1) *piggybacking* different requests as a means of decreasing communication; (2) *early misspeculation detection* by leveraging the existing cache mechanisms to avoid the expensive misspeculation checks when possible; and (3) a *fast recovery* scheme to speedup re-computations by leveraging the contents of the existing speculation buffer.

5.2 Interplay Between Distributed Software Speculation and Distributed Caching

In this section we begin by motivating the need for using speculation to exploit dynamic parallelism present in irregular applications and then describe how speculation is implemented in a DSM based system that employs distributed caching. Finally, we present results of an experimental study that demonstrates the interplay between distributed caching and distributed speculation. For non-speculative distributed data parallel applications, larger caches deliver higher performance as larger caches lead to higher cache hit rate and thus a reduction in long latency fetch operations. Our study shows that in the presence of speculation while larger caches can result in higher cache hit rates, they also lead to a greater likelihood that at least some of the values found in caches are stale. This leads to a higher misspeculation rate and thus an overall degradation in performance.

5.2.1 Distributed Caching and Speculation Protocol

For this study, we have built a directory-based caching DSM system modeled on the recently proposed system by Dash et. al. [30]: the input data is partitioned among the nodes in the cluster and each object is assigned a *home node.* The cache contains copies of remote objects while the directory keeps track of all the objects in the cache and some state information associated with the objects. The cache is a directory-based, distributed write-through cache. Any object in the DSM has a state that is either *Uncached* or *Shared* as shown in Figure 5.1a. Any shared object supports the following four basic operations: Read, Write, Invalidate and Evict. For each shared object o, the directory maintains information about the machine m_i that has cached object o. Specifically, if machine m_i requests a read/write of object o, then the directory marks it shared and adds m_i to the list of machines that has cache object o. The directory entry for o corresponding to m_i is appropriately updated to reflect evictions and invalidate messages.

The speculation we use is similar to that presented by Dash et. al [30]. Object versioning is used to enable speculation: each object is associated with a version number which is used to detect dependency violations. The software speculation mechanism consists of three phases: (1) Compute, (2) Misspeculation detection, and (3) Update. In the *compute* phase, a private, thread-local *Speculative Buffer* is used to privately cache all the data required to perform the computation independent of all other computations. The speculative buffer also holds the version numbers associated with the objects used in the computation. Any updates to be performed by the computation are made only to the thread-local speculative buffer.

Next, in the *misspeculation detection* phase, the data used in the compute phase is verified to be current. This is achieved by *lock* on all the data in the speculative buffer to ensure an atomic update. Then, the version number of the objects in the speculative cache is checked against those of the authoritative copies. If any version numbers *mismatch*, the computation is deemed misspeculated and the computation aborts. If there is no misspeculation, then the update is performed. From the above description, we observe that to support distributed speculation, a distributed cache should support: (1) versioning for objects, and (2) read/write-locking of objects. The misspeculation detection and update, henceforth collectively referred to as *Speculative Commit*, are summarized in Algorithm 5, which attempts to atomically *commit* object O using the associated speculative buffer sb which contains the data used to compute object O.

Figures 5.1a and 5.1b show the cache and directory, while Figure 5.1c shows the states of the an object o in the speculative buffer. The speculation-specific parts are in **blue**.

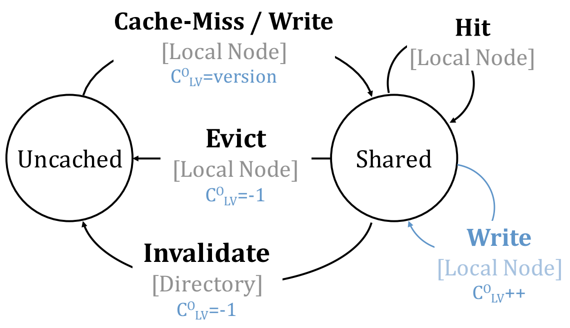
Figure 5.1a includes a *Cache Local Version*, C^{o}_{LV} for each object o, representing the version of the object in the cached. On a cache miss, when object o is read into the

Algorithm 5: Speculative Commit.

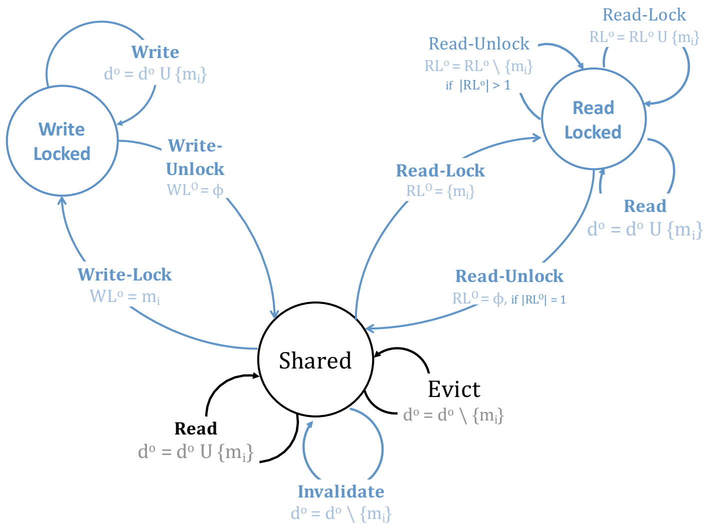
```
1  SpeculationBuffer sb;
2  for int i=0; i<sb.size(); i++ do                                  /* Lock */
3      lock(sb[i]);
4  for int i=0; i<sb.size(); i++ do
5      Object o = remote-fetch(sb[i]);                               /* Ver chk */
6      if o.version != sb[i].version then
7          for int j=0; j<i; j++ do
8              unlock(sb[j]);
9          return failure;
10 Write(O)                                      /* No misspeculation:  commit */
11 for int i=0; i<sb.size(); i++ do                                  /* Unlock */
12     unlock(sb[i]);
13 return success;
```

local cache, C^o_{LV} is set to the version of the remote object. When invalidated or on evict, C^o_{LV} is set to -1. The directory is extended to support read and write locks. The directory now has two additional states: Read-Locked and Write-Locked. WL^o is the machine m_i that currently holds the write lock on object o and RL^o is the *set* of machines m_i that have currently requested read locks on the object. Observe that the directory protocol allows single writer and multiple readers.

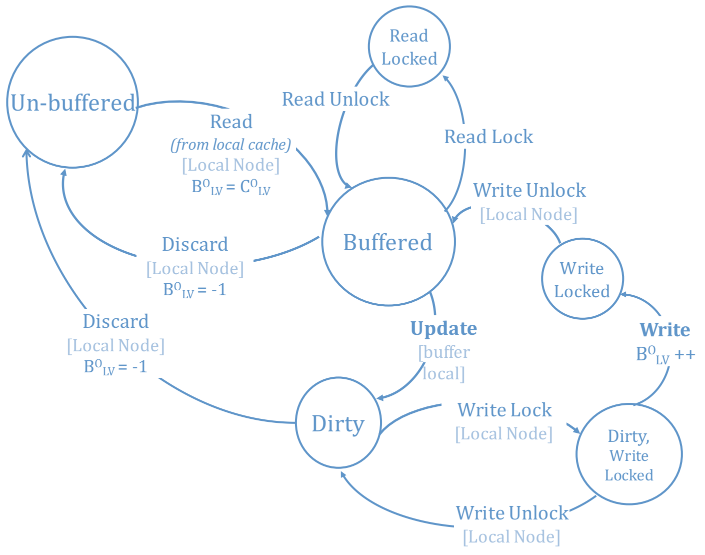
To further aid protocol analysis, Figure 5.1c presents the states of objects in the *speculative buffer*. As with any cache, an object can be *Buffered* or *Unbuffered*; we use the phrase buffer to distinguish the speculative buffer from the machine cache. Buffered objects can be *Dirty* (on update) and dirty objects can be write-locked (just before update). On successful update, the dirty object is simply marked buffered. Buffered objects in the speculative buffer can also be read-locked (during speculative commit). After successful commit or on abort, objects in the speculation buffer are marked unbuffered and discarded.



(a) States for object in cache with speculation.



(b) States for objects in the directory with speculation.



(c) States for objects in speculative buffer with speculation.

Figure 5.1: Cache, directory and speculative buffer state diagrams for speculation.

5.2.2 Cache Size vs. Misspeculation Rate

We now study how the performance of distributed speculation on a DSM with caching can degrade with larger caches. We evaluate our system on various graph mining and analytics applications on various real world inputs.

Experimental Setup: Table 5.1 lists the various applications used in this evaluation. Graph Coloring (GC) is an algorithm that assigns 'colors' to nodes in the graph such that no two neighbors have the same color. As the name suggests, Connected components (CC) computes the connected subgraphs of a graph. PageRank (PR) computes the 'rank' of a webpage represented by a graph vertex based on the rank of its neighbors. Community detection is similar to CC, and SSSP computes the shortest path to each vertex in the input graph from a given source.

Applications
Graph Coloring (GC)
Connected Components (CC)
Community Detection (CD)
Single-Source Shortest Path (SP)
PageRank (PR)

Table 5.1: Applications

Table 5.2 summarizes the real world inputs from the Stanford Network Analysis Project (SNAP) [78], University of Koblenz-–Landau's Konect Network Collection [72] and The University of Florida Sparse Matrix Collection [33] along with their sizes. $|V|$ is the number of vertices and $|E|$ is the number of edges in the input. WordNet [72] is a network between words in the Wordnet database with words being the vertices and relationships between words, the edges. Gowalla [72] is a graph from the now defunct Gowalla social network, with nodes representing people and edges representing friendships. DBLP [72] is a DBLP bibliography database's co-authorship network with authors as vertices and an edge between authors with joint papers. Amazon [78] is a co-purchase graph with vertices representing products and edges representing a co-purchase. These three inputs are power law graphs with many high degree vertices with up to a degree of 14,730. DielFilterV3Real [33] is a graph representation of a sparse, high-order finite element matrix. Flan1565 [33] is a graph representation of a steel flange. The last two inputs are sparse matrices with only low degree nodes, with a maximum degree of 10.

Input	$\lvert V\rvert$	$\lvert E\rvert$
WordNet (WN)	146,005	656,999
Gowalla (GW)	196,591	950,327
DBLP (DB)	317,080	1,049,866
Amazon (AZ)	400,727	3,200,440
DielFilterV3Real (DF)	1,102,824	89,306,020
Flan1565 (FL)	1,564,794	114,165,372

Table 5.2: Real-world inputs

We now evaluate the protocol on an 8-node cluster running CentOS 6.3. The nodes in the cluster are connected to a Mellanox Infiniband switch. In addition to the speculation, to avoid wasteful computation we employ *activations* [82]. In the first iteration, every input data object is scheduled for computation. In subsequent iterations, only those that could potentially require an update are scheduled for computation. For example in a graph, if a particular vertex is updated in the current iteration, all of its immediate neighbors are *activated* for the next iteration, as they depend on a changed vertex. As a final enhancement, the speculative commit is attempted only if the value of the object changes.

Cache%		GW		AZ	
		Time	MR	Time	MR
GC	1	646	2.49	135	11.24
	5	498	1.04	102	9.69
	10	388	0.63	87	6.37
	20	516	0.85	108	8.30
CC	1	514	2.38	112	10.63
	5	449	1.22	111	10.08
	10	326	0.72	112	9.69
	20	530	1.11	119	10.59
CD	1	1187	2.12	101	9.47
	5	721	0.95	88	8.82
	10	668	0.52	84	9.35
	20	749	0.77	88	9.50
SP	1	7	0.00	119	6.39
	5	7	0.00	77	5.59
	10	6	0.00	77	5.44
	20	6	0.00	90	6.58
PR	1	1241	2.27	456	9.42
	5	1054	1.14	457	9.59
	10	1106	1.74	402	9.45
	20	1225	1.19	425	9.77

Table 5.3: Effect of cache size on speculation on an 8 node cluster. Execution time in seconds and misspeculation rate (MR) with varying cache sizes.

Peformance of Distributed Speculation on a Caching DSM: Table 5.3 shows the total running times for distributed speculation on the caching DSM described above with varying cache sizes along with their misspeculation rates (MR). MR is computed as the ratio of misspeculations over cache hits. The general trend here is a speedup from 1% cache to 10% cache and then the speculation slows down at 20% cache size. That is, even though the number of remote fetches are decreased with larger cache size, the increase in misspeculative aborts degrades the performance much more than any potential savings from the cache. The highlighted values are the data of interest showing that in many cases the distributed speculation performs best with smaller caches. In most cases, the fastest configuration also corresponds to lowest MR. In other cases, the benefit from using cached values out weighs the cost of misspeculation, leading to faster execution with only slightly larger MR than the minimum for that application-input combination. In a few cases, the speculation is insensitive to cache sizes: for example, all executions on the FL input, which is a essentially a sparse graph with max degree of about 10. In this case, the misspeculations are fewer with no impact from the cache.

5.3 Optimizing Distributed Speculation

Optimizations can be found at various layers of the protocol. We now look at various optimizations to the speculation protocol, leveraging the state information already captured in the cache and directory structures.

5.3.1 Piggybacking Version

To begin with, consider the Line 3 and Line 5 of Algorithm 5. We can save on communication by combining the lock request and fetch for version checking into a single request: we can piggyback the version of the requested object in the lock request message, thus saving about 50% of communication requests over the baseline commit. The remote lock request will return failure without acquiring a lock if versions don't match. Algorithm 6 summarizes the *Speculative Commit* protocol with this optimization.

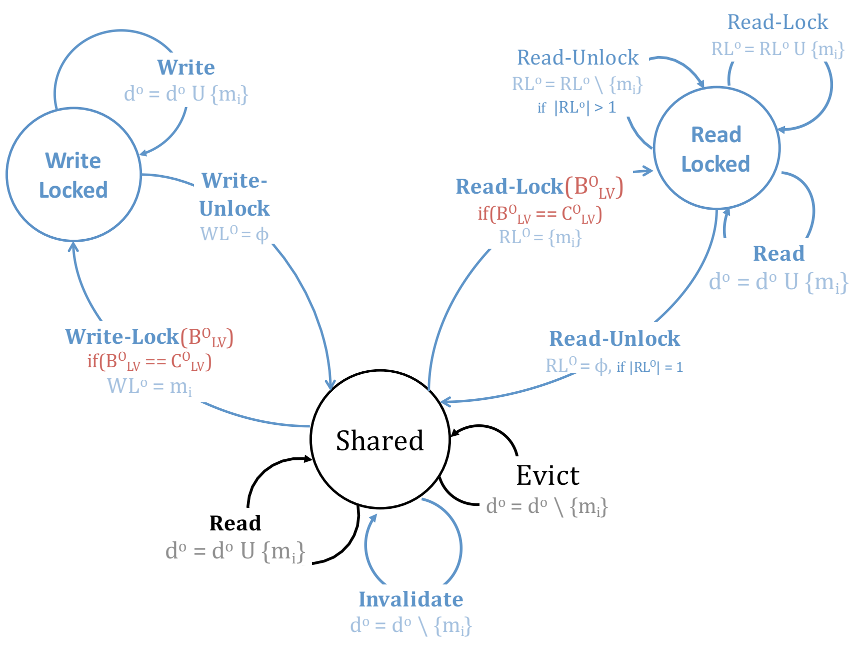
Algorithm 6: Speculative Commit with Piggybacking.

```
1  SpeculativeBuffer sb;
2  for int i=0; i<sb.size(); i++ do                          /* Lock+Check */
3      if ! lock(sb[i], sb[i].version) then
4          for int j=0; j<i; j++ do
5              unlock(sb[j]);
6          return failure;
7  Write($O$)                                 /* No misspeculation: commit */
8  for int i=0; i<sb.size(); i++ do                              /* Unlock */
9      unlock(sb[i]);
10 return success;
```

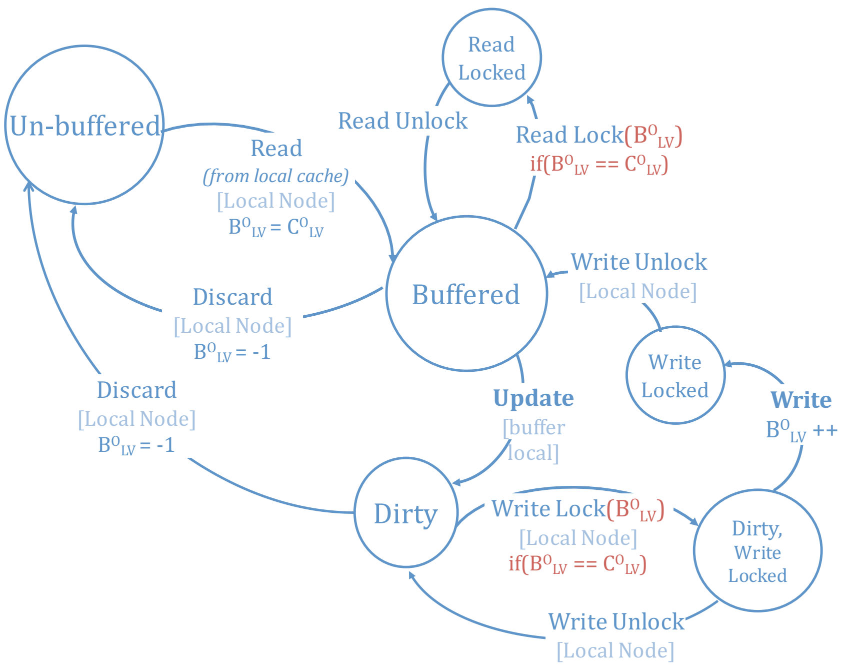
Figure 5.2 shows the extensions to the baseline speculation in Figure 5.1 in **red**. B^o_{LV} refers to the *Buffer Local Version* of object o in the speculative buffer. The lock mechanism is modified to fail if $B^o_{LV}! = C^o_{LV}$. Observe that there are no changes to the cache itself; only the directory and speculative buffer need modification.

5.3.2 Early Misspeculation Detection

As a next optimization, we leverage the fact that the cache receives invalidate messages. Given that an object is present in the cache of a given machine *only* if it was read by a compute thread on that machine, we can extend the cache to track which threads on the local machine are *actively* using each object. Anytime an object is invalidated, it indicates that all the threads that have read and are using the invalidated object have definitely misspeculated. This allows us to completely bypass the expensive speculation commit attempt, thereby giving us better performance. Notice that this optimization only requires changes to the cache.



(a) States for objects in the directory with piggybacking.



(b) States for objects in speculative buffer with piggybacking.

Figure 5.2: State Diagrams for Speculation with Piggybacking.

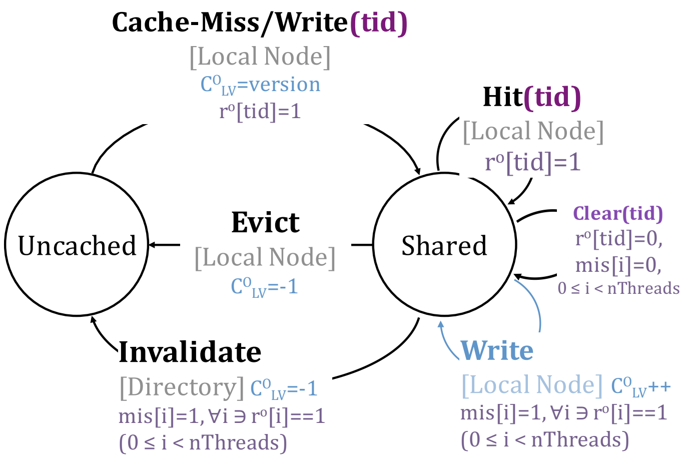


Figure 5.3: States for objects in cache with early misspeculation detection.

Figure 5.3 shows the modifications over the basic speculation from Figure 5.1 in **purple**. For each compute thread *tid* that reads object o, we set the r^o flag. To keep track of misspeculation, we use an array of booleans, one for each thread: $mis[]$. If thread *tid* writes object o or the cache receives an invalidate message, the cache sets $mis[i]$, for all threads i that have read o; i.e., all threads i that have $r^o[i]$ set. In addition, we introduce a $Clear(tid)$ method which clears the read flags for all objects that thread *tid* had read. This method is invoked on abort and after successful commit.

5.3.3 Fast Recovery

We now explore an optimization to help speed up re-computations that arise from misspeculations. Given that during misspeculation detection, the speculative buffer knows which object(s) have been misspeculated, the basic idea is the following: if only a subset of the objects in the speculative buffer have resulted in misspeculation, then the re-computation could hedge that other objects *may* not have changed and only re-fetch the misspeculated objects, while unchanged objects are read from the speculative buffer.

The expectation here is to gain speedups from fewer fetches. A further optimization: our framework computes and updates objects using some ordering on input data. In the speculative commit phase, when acquiring locks for objects in the speculative buffer, the lock requests on objects are also ordered using the same ordering to prevent deadlocking. If we use the simplistic fast-recovery mechanism just described above, we observed that as soon as we encounter one misspeculated value, the probability that many of the following objects are also misspeculated is very high. This is expected since the computations are ordered using the same mechanism as the locks and therefore if a misspeculation has occurred on o_i since it was recently updated, the there is a good chance that o_j, $j > i$ also have been updated, and therefore potentially misspeculated. Hence, we further optimize by fetching all the objects following a misspeculated object from the machine cache rather than re-use from the speculative buffer.

Algorithm 7: Speculative Commit with Fast Recovery.

```
1  SpeculativeBuffer sb;                                  /* Ordered buffer */
2  fail = false;
3  int i=0 for i=0; i<sb.size(); i++ do                   /* Lock */
4      if ! lock(sb[i], sb[i].version) then
5          sb[i].changed = true;
6          fail = true;
7          break;
8  if fail then
9      for int j=0 j<i; j++ do
10         unlock(sb[j]);                                 /* Release if locked */
11     return failure;
12 Write(O)                                   /* No misspeculation:  commit */
13 for int i=0; i<sb.size(); i++ do                       /* Unlock */
14     unlock(sb[i]);
15 return success;
```

Algorithm 8: Recomputation with Fast Recovery.

```
1  SpeculativeBuffer sb;                          /* Ordered buffer */
2  int i=0;
3  for i=0; i<sb.size(); i++ do
4      if ! sb[i].changed then                    /* Unchanged */
5          O = compute(O, sb[i]);
6      else
7          break;
8  for int j=i; j<sb.size(); j++ do
9      Object o = fetch(sb[i]);/* Maybe changed                    */
10     O = compute(O, o);
```

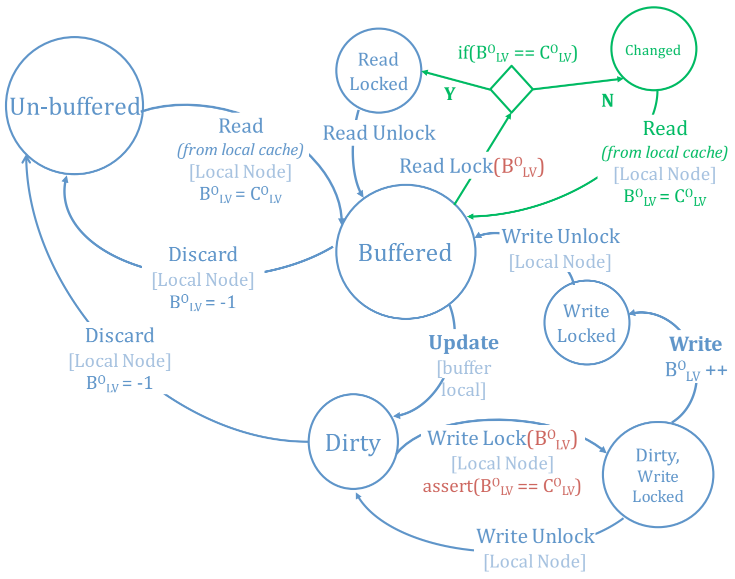

Un-buffered
Read Locked
Changed
if(B^O_{LV} == C^O_{LV})
Y
N
Read (from local cache) [Local Node] $B^O_{LV} = C^O_{LV}$
Read Unlock
Read Lock(B^O_{LV})
Read (from local cache) [Local Node] $B^O_{LV} = C^O_{LV}$
Buffered
Write Unlock [Local Node]
Discard [Local Node] $B^O_{LV} = -1$
Write Locked
Discard [Local Node] $B^O_{LV} = -1$
Update [buffer local]
Write B^O_{LV} ++
Dirty
Write Lock(B^O_{LV}) [Local Node] assert(B^O_{LV} == C^O_{LV})
Dirty, Write Locked
Write Unlock [Local Node]

Figure 5.4: States for objects in speculative buffer with fast recovery.

Algorithms 7 and 8 summarize the changes needed to support misspeculation and recomputation with fast recovery. For fast recovery, only the speculative buffer sub-system needs modification. Figure 5.4 shows the extensions to support fast recovery in **green**. We introduce a new state for buffered objects in the speculative buffer called *Changed* that tracks misspeculated objects. During recompute, we read the unchanged objects from the speculative buffer and the changed objects from the cache. Buffered object are moved to the changed state depending on the success of the read-lock: if the lock fails since the buffered version is outdated, the object is marked changed.

5.4 Evaluation of Optimizations

The experiments were performed on a cluster of 8 machines running CentOS 6.3, each with 32 cores and 64GB main memory with a Mellanox Infiniband interconnect switch. For the DSM we used the directory-based, write-through, strict consistency caching protocol from the ASPIRE system [129], with the extensions described above to support speculation.

5.4.1 Overall Speedups from Optimizations

	GW		AZ		DF	
	Speedup	ΔMR	Speedup	ΔMR	Speedup	ΔMR
GC	2.24	-1.09	2.05	-4.45	1.07	0.06
CC	1.71	-0.20	1.59	-2.73	1.09	0.09
CD	1.88	-0.84	1.38	-5.01	1.17	0.00
SP	1.00	0.00	1.18	0.38	1.01	-0.01
PR	1.74	-0.37	1.67	-2.73	1.05	0.08
	FL		**WN**		**DB**	
	Speedup	ΔMR	Speedup	ΔMR	Speedup	ΔMR
GC	1.06	-0.06	1.12	-0.44	1.17	-0.51
CC	1.03	-0.09	1.12	-0.45	1.10	-0.10
CD	1.00	0.00	1.06	-0.17	1.07	-0.07
SP	1.00	0.00	1.00	0.00	1.00	0.00
PR	1.20	-0.21	1.12	-0.33	1.04	0.48

Table 5.4: Best speedups and change in MR (ΔMR) of optimized speculation over baseline.

Table 5.4 presents the best achievable performance for various applications over the baseline from Table 5.3. Observe that these optimizations provide up to 2.25x speedup. Notice that except for 5 cases, the speedups are accompanied by a decrease in MR.

Cache%		GW		AZ		DF		FL		WN		DB	
		Speedup	ΔMR	Speedup	ΔMR	Speedup	ΔMR	Speedup	ΔMR	Speedup	ΔMR	Speedup	ΔMR
GC	1	2.24	-1.09	2.05	-4.45	1.02	0.08	1.04	-0.02	1.12	-0.44	1.11	-0.47
	5	1.72	-0.35	1.79	-3.50	1.07	0.06	1.06	-0.06	1.07	-0.46	1.17	-0.51
	10	1.41	-0.13	2.04	-3.15	1.04	0.09	1.04	-0.04	1.09	-0.30	1.12	-0.46
	20	1.84	-0.36	1.67	-1.76	1.05	0.06	1.03	0.01	1.07	-0.68	1.08	-0.47
CC	1	1.62	-0.56	1.47	-2.76	1.05	0.14	0.99	-0.11	1.02	-0.17	1.10	-0.10
	5	1.57	-0.47	1.56	-2.42	1.04	0.14	1.03	-0.14	1.04	-0.63	1.00	0.15
	10	1.13	-0.56	1.45	-1.38	0.99	0.09	1.02	-0.11	1.12	-0.45	1.00	0.15
	20	1.71	-0.20	1.59	-2.73	1.09	0.09	1.03	-0.09	1.04	-0.44	1.09	-0.22
CD	1	1.88	-0.84	1.28	-2.61	1.17	0.00	1.00	0.00	0.99	-0.36	1.06	-0.35
	5	1.10	-0.21	1.26	-2.91	1.00	0.00	1.00	0.00	1.06	-0.17	1.03	-0.08
	10	1.41	-0.24	1.24	-4.34	1.00	0.00	1.00	0.00	1.01	-0.06	1.04	-0.28
	20	1.22	-0.08	1.38	-5.01	1.00	0.00	1.00	0.00	0.90	-0.85	1.07	-0.07
SP	1	1.00	0.00	1.17	0.99	0.99	-0.04	1.00	-0.00	1.00	0.00	1.00	0.00
	5	1.00	0.00	0.79	2.30	1.01	-0.01	1.00	-0.00	1.00	0.00	1.00	0.00
	10	1.00	0.00	1.00	1.29	1.00	0.00	1.00	-0.00	0.67	0.00	1.00	0.00
	20	1.00	0.00	1.18	0.38	0.94	0.09	0.86	-0.00	0.67	0.00	1.00	0.00
PR	1	1.62	-0.15	1.44	-1.59	1.05	0.08	1.10	-0.23	1.12	-0.33	1.04	0.48
	5	1.31	-0.50	1.6	-2.73	0.98	0.12	1.20	-0.21	1.12	-0.38	0.97	0.56
	10	1.46	-0.26	1.49	-2.26	1.02	0.11	0.96	-0.17	1.09	-1.04	1.00	0.54
	20	1.74	-0.37	1.50	-2.65	1.02	0.11	1.10	-0.26	1.11	-0.36	1.00	0.35

Table 5.5: Speedups and the change misspeculation rate (ΔMR) for optimized speculation with varying cache sizes.

Table 5.5 shows the speedups and ΔMR for various cache sizes over the baseline data presented in Table 5.3. Notice that compared to the baseline many more configurations are able to achieve their best speedups at *larger* cache sizes. The highlighted data points out configurations with the maximum speedups from these optimizations. Observe that the best speedups are usually achieved when the decrease in MR is the largest or the increase in MR is the smallest. That is, these optimizations mostly decrease misspeculation as expected. For the configuration where the speedups are gained with an increase in MR, observe that the gains are only marginal, as can be seen in the highlighted rows for DF and the highlighted PR with DB input.

5.4.2 Combining Optimizations

Next, we present the speedups obtained from the individual optimizations and their combinations to study their specific share of contribution to the speedups. Table 5.6 shows the speedups obtained from *Piggybacking*, *Early misspeculation Detection*, *Fast Recovery* and their pair-wise combinations. Considering the individual optimizations in the top half of Table 5.6, we see that the maximum benefits are from *piggybacking*, which focuses on decreasing the communication in the misspeculation phase. Piggybacking shows a maximum of $\approx$2x speedup, early misspeculation detection a maximum of $\approx$1.8x speedup while fast recovery shows a maximum speedup of $\approx$1.5x. In these experiments, piggybacking and early misspeculation detection are helpful in many cases, while fast recovery shows speedups in fewer cases. Fast recovery tends to be less effective since the recomputation phase may incorrectly speculate that many objects in the speculative buffer are unchanged. In such cases, the fast recovery optimization could potentially increase the aborts with lesser benefit.

		GW	AZ	DF	FL	WN	DB
Piggybacking	GC	1.72	1.99	1.04	1.08	1.17	1.08
	CC	1.44	1.61	1.08	1.05	1.07	1.10
	CD	1.74	1.52	1.00	1.00	1.01	1.04
	SP	1.00	1.32	1.08	1.00	1.00	1.00
	PR	1.61	1.72	1.06	1.14	1.09	1.08
Early Misspec. Detection	GC	1.56	1.59	1.06	1.06	1.10	1.08
	CC	1.17	1.30	1.12	1.02	1.02	1.10
	CD	1.78	1.22	1.00	1.00	1.08	1.07
	SP	1.00	1.19	1.05	1.17	1.00	1.00
	PR	1.32	1.07	1.01	1.22	1.06	1.04
Fast Recovery	GC	1.19	1.48	0.98	1.01	1.00	1.00
	CC	1.11	1.02	1.28	1.04	1.00	1.04
	CD	1.21	1.09	1.17	1.14	1.07	1.00
	SP	1.20	1.14	1.01	1.17	1.00	1.00
	PR	1.26	1.06	1.00	1.04	1.00	0.99

Table 5.6: Best speedups from individual optimizations over baseline speculation.

		GW	AZ	DF	FL	WN	DB
Piggybacking + Early Misspec. Detection	GC	1.77	2.14	1.07	1.07	1.14	1.08
	CC	1.62	1.72	1.09	1.05	1.09	1.10
	CD	2.08	1.42	1.17	1.14	0.97	1.12
	SP	1.20	1.50	1.11	1.00	1.00	1.00
	PR	1.71	1.70	1.07	1.04	1.10	1.09
Early Misspec. Detection + Fast Recovery	GC	1.68	1.44	0.99	1.02	1.06	1.07
	CC	1.36	1.32	1.09	1.05	1.02	1.06
	CD	1.57	1.12	1.17	1.14	1.00	1.07
	SP	1.20	1.07	1.01	1.00	1.00	1.00
	PR	1.27	1.07	1.03	1.20	1.03	1.00
Piggybacking + Fast Recovery	GC	1.62	1.96	1.07	1.02	1.12	1.08
	CC	1.64	1.61	1.08	1.03	1.09	1.06
	CD	1.94	1.31	1.00	1.00	1.01	1.07
	SP	1.17	1.29	1.01	1.00	1.00	1.00
	PR	1.79	1.69	1.02	1.10	1.08	1.03

Table 5.7: Best speedups from pairwise combination of optimizations over baseline speculation.

As can be expected from the above discussion, the piggybacking + early misspeculation detection combination is the most effective pair-wise combination, giving a maximum speedup of 2.14x. Finally observe that compared to the pair-wise combinations, the synergy between all three optimizations as shown in Table 5.4 is less, but greater when present. For example, Graphcoloring, Connected components and PageRank with the GW input all perform better than any of the individual or pair-wise optimizations. These gains obtained from the pairwise combinations of these optimizations is shown in Table 5.7.

5.4.3 Integration with Infinimem

We now present the results of integrating the object-centric *InfiniMem* library into the DSM to further support scale-up on individual machines in the cluster. In this experiment, the DSM was allocated memory to hold only 75% of the AZ input that was partitioned and assigned to it. The remaining 25% of the local data is stored on the disk. As can be expected, this feature would slightly slowdown the performance of distributed speculation; Table 5.8 shows the percentage slowdown. Observe that the overhead results in an average slowdown of only 5.3%. In this case, the overhead is from individual object reads and writes, which arise from random accesses generated by the applications.

Application	Overhead (%)
GC	5.8
CC	7.1
CD	8.2
SP	2.5
PR	2.8

Table 5.8: Overhead from using *InfiniMem* to spill data that does not fit in allocated memory is shown here as a percentage of slowdown over the version without *InfiniMem* for AZ input.

Finally, we compare the performace of size oblivious distributed software speculation with the data in Table 1.1. Table 5.9 shows the parallel input load/output store, compute and size oblivious I/O via *InfiniMem* for various applications.

App	Load/Store in seconds	Compute in seconds	*InfiniMem* I/O in seconds	Total Time in seconds	Speedup (over Table 1.1)
GC	2	45.8	5	52.8	6.6
CC	2	62.3	8	72.3	8.3
CD	2	56.3	7	65.3	10.2
SP	2	68.6	7	77.6	8.7
PR	3	265.4	4	272.4	3.6

Table 5.9: Breakdown of total time into input, size oblivious IO and compute for distributed software speculation with *InfiniMem*.

Compared to the runs from Table 1.1, we see overall speedups between 3.6x – 10x, with an average speedup of ∼7.5x. Parallel I/O speedups up input loading/output storage more than 75% in this experiment. Prallelizing the program on a cluster results in an inrease in fraction of time spent in computation rather than on intermediate I/O. Earlier, most of the time (∼95%) was spent intermediate I/O, while most of the time (∼88%) is now spent on computation and very little (∼8%) on intermediate I/O. This reaffirms the motivation for size oblivious programming of clusters for irregular applications.

5.5 Summary

This chapter presented the protocols for distributed software speculation on a caching DSM as a framework for analyzing and optimizing the distributed speculative commit protocol. We then identified the following three optimizations to speedup or bypass the expensive misspeculation detection and to speedup the recomputation after misspeculation has occurred: (1) piggybacking object version numbers on lock requests to decrease the communication cost of the misspeculation check; (2) early misspeculation detection leverages the information contained in the cache to entirely avoid the expensive speculative commit; and (3) fast recovery to speed up the re-computations resulting from misspeculative aborts, by fetching only potentially changed objects in the speculative buffer. These optimizations provide between 1.32x to 2.24x speedups on various graph processing applications. Finally, we showed that *InfiniMem* can scale-up individual nodes on the cluster by spilling data that does not fit in local memory with an average overhead of just 5.3%.

Chapter 6

Related Work

This chapter summarizes various research in domains and problems addressed by this thesis. We first summarize prior programming solutions for big data problems on DSMs and single machines. Next, we address various attempts at size-oblivious programming including distributed computing solutions. Finally, we summarize work on latency tolerance mechanisms and their impact on speculation.

6.1 Programming Interface

First, we present related research on programming systems with large datasets, followed by work on programming speculation.

6.1.1 Programming Large Data

The earliest solutions to programming large distributed systems involved manual message passing, which was later standardized via the Message Passing Interface (MPI) [44].

MPI standardized the communication primitives and provided a unified, efficient framework to pack/unpack, serialize/deserialize many different types of data structures portably across varying machine types [93, 48]. A related approach to keep programming simple on distributed platforms was the development of the notion of Distributed Shared Memory (DSM). Programs are written using the familiar shared memory paradigm and the DSM runtime employs a distributed coherent cache. The data is partitioned across machines in the cluster. The caching protocol on the DMS system uses message passing or similar techniques to transparently fetch and store data needed by computations on various machines in the cluster.

DSMs are not the only programming/memory model for distributed parallel programming. MapReduce [34] is a popular programming model that consists of two distinct phases of computation: map and reduce. The map operation distributes work across the cluster while the reduce operation aggregates the results from the across cluster. However, MapReduce cannot be applied to every program that can be parallelized, thereby limiting it to a smaller set of applications than possible with DSM. Another approach is to use distributed memory (DM) (in contrast to distributed *shared* memory (DSM)) model. With DM, the programmer has to explicitly manage and move data between various compute nodes. HipG [68] is an example of a graph processing framework that uses DM. HipG contains DM extensions similar to those proposed in SpiceC [40] and works in two phases, like MapReduce. The Message Passing Interface (MPI) [44] is a popular DM programming model. Compared to DSM based systems, the programmer burden in programming for these systems is very high. For example, with the exception of transferring atomic data types and their arrays, simply creating the wrappers to allow serialized communication of incrementally complex data structures can be daunting enough to deter the use of this system for larger programs. Examples of some recent Distributed Memory (DM) systems are Hadoop [118] and Pregel [84]. Here, programming the system to bring in required data is a major task. Such systems usually employ an underlying Distributed File System (DFS) like the Hadoop Distributed File System (HDFS) [119] or the Goole File System (GFS) [45]. Specialized systems like GraphLab [82], GraphX [131] etc. support efficient distributed processing of graph applications. Our framework is designed to be general purpose and works with arbitrary data types.

Our approach simply requires the programmer to identify the data collections that can grow large using the `large` keyword. Such data collections are automatically partitioned and distributed across the cluster. In addition, data that cannot fit in available memory is transparently and automatically spilled to available disk using out *InfiniMem* library.

6.1.2 Programming Speculation

LRPD was amongst the earliest practical implementations of software speculation technique [105, 104]. The compiler was designed to look for loops that require speculation and then parallelized them as a `doall` loop with speculation. In our approach, the programmer annotates loops which need to be speculatively parallelized – this speeds up the compilation itself compared to the compiler searching through all loops in the program. The savings can be especially significant for very large programs. Our approach is similar to that taken by SpiceC [40], which employed various incarnations of the `#pragma speculate` pre-processor directives around loops, with each variation addressing a different type of parallelism.

However, all this was only in the context of single, shared memory machines. Programming language support for `doall` speculative parallelization is not present in previous and existing DSM systems and big data frameworks. In our approach, a simple `speculate` annotation on a loop is transformed into a distributed, speculative `doall` loop.

6.2 Size Oblivious Programming

The closest file organization to that used by *InfiniMem* and illustrated in Figure 3.6 is the B+ tree representation used in database systems. The primary differences in our design are the following: (1) *InfiniMem* uses a flat organization, with at most one level index for variable sized data. (2) *InfiniMem* provides O(1) time I/O operations for random access while the B+ trees require O(log n) time. A related approach uses Sorted String Tables (SST). For example, LevelDB [51] and RocksDB [50] use this organization. LevelDB for example, stores files in various 'levels', combining files into the next larger level file when it grows too large. The disadvantage of this approach is that the entire level file is required to be held in memory. In addition, searching for items in the level file requires sequential scans. Our approach does not require the entire data file in memory and can locate items randomly, in at most two logical seeks.

6.2.1 Out-of-core Computations

In this work, we enable applications with very large input data sets to efficiently run on a single multicore machine, with minimal programming effort. The design of the *InfiniMem* transparently enables large datasets become disk-resident while common out-of-core algorithms [28, 71, 127] *explicitly* do this. As demonstrated with shards, it should be easy to program these techniques with *InfiniMem*.

6.2.2 Processing on a Single Machine

Traditional approaches to large-scale data processing on a single machine involve using machines with very large amounts of memory, while *InfiniMem* does not have that limitation. Examples include Ligra [117], Galois [99], BGL [120], MTGL [11], Spark [133] etc. FlashGraph [135] is a semi-external memory graph processing framework and requires enough memory to hold all the edgelists; *InfiniMem* has no such memory requirements.

GraphChi [75] recently proposed the Parallel Sliding Window model based on *sharded* inputs. Shard format enables a complete subgraph to be loaded in memory, thus avoiding random accesses. GraphChi is designed for and works very well with algorithms that depend on static scheduling. *InfiniMem* is a general-purpose size-oblivious programming framework and recognizes the need for sequential/batched and random input for fixed and variable sized data and provides simple APIs to handle all forms of I/O for rapid prototyping.

6.3 Distributed Shared Memory Software Systems

Distributed Shared Memory or DSM is a prominent choice of memory models for programming distributed systems. In essence, this software layer superimposes an abstraction of shared memory over the physical memories from multiple machines in a cluster. This allows programmers to write applications using the familiar abstraction of shared memory systems. Additionally, the scalability of clusters coupled with the shared memory abstraction makes DSMs an attractive option for scalable distributed computing without resorting to specialized programming models or frameworks.

6.3.1 Latency Tolerance Mechanisms

Dynamic data prefetching in the context of distributed systems has also been studied [29, 80] in various research and engineering communities. Our work differs by providing a runtime that monitors dynamic parameters and dynamically turns on/off prefetching as needed. Further, our designation of separate threads on each machine specifically for prefetching allows us to dynamically scale prefetching on demand. This further allows for optimal allocation of physical CPU cores to computation and communication needs of the application. There are other strategies based on Markov models [57] etc., that do not need any input from the user, but those models are not the primary focus of this evaluation.

Traditional clusters were built from commodity single core CPUs and the DSMs proposed for those systems were not designed to exploit the presence of multiple cores. For instance, systems like ORCA, Shasta, TreadMarks and Emerald [8, 58, 4, 112] were all successful DSM systems for clusters of uniprocessor machines. Our work is focused not on the DSMs *perse*, but rather on novel approaches to exploit new opportunities afforded by multi-core machines. Specifically, we explore the dynamic balance between communication and computation for speculative and asynchronous applications on DSMs. Prior systems do not provide explicit support for speculative or asynchronous parallelism, like we do. Finally, to the best of our knowledge, none of the older systems monitored dynamic, runtime characteristics of data and applications to automatically *adapt communication and computation* for optimal performance. To tolerate latencies, prior work has explored lazy vs. eager release consistency models [4]. Other work has looked at dynamically adapting between single and multiple writer protocols [89, 3] or adapting to dynamic sharing patterns, which again relies on some release consistency model [89] in the context of regular applications. This work focusses on irregular applications; for speculative applications, we rely on the SpiceC [40] model, which is a lazy release, multiple-writer protocol. Asynchronous applications by definition require no strict consistency models; we use the multiple writer model for both speculative and asynchronous applications, with primary focus on ABC^2: Adaptively Balacing Communication and Computation.

6.3.2 Distributed Software Speculation

Prior efforts for achieving distributed speculation in a DSM based cluster explore alternate approaches to speculation. For example, executing multiple processes with the same input while speculating that one or more instances may either fail or complete early is a speculative approach to robustness, as in Hadoop/Hive. In master-slave speculation, multiple master threads compute different approximate versions of the program while the slave threads validate the computation, aborting the invalid masters [136]. A similar approach is taken by algorithmic speculation where multiple algorithms for the same problem are speculatively executed in parallel and the most appropriate result is accepted [107]. A more recent work has considered integrating caching with symbolic prefetching to support distributed speculation to enable doall parallelism in irregular applications [30].

In this work, we first show that speculation on DSMs with caching can slow down with larger caches due to an increase in misspeculation rates from reading potentially stale values from the cache. We then propose three optimizations that leverage the caching protocol to decrease the cost of communication, misspeculation check and speedup misspeculated recomputations by combining requests, leveraging information present in the cache and the thread-private speculation buffer etc.

The baseline speculation and caching DSM protocol used in this work is based on the system recently presented by Dash et. al. [30]. It uses execution model based on SpiceC's [40] implicit private copying and explicit commits; the speculation works by making private copies to enable compute isolation and speculative execution following LRPD's original proposal for shared-memory software speculation [105], which also helps with speculative execution in a distributed setting. Like LRPD, our approach also speculatively executes all iterations of the loop in a 'doall' manner. The distributed misspeculation check is similar to D-BOP's distributed validation [52]. In addition, we present the protocol for distributed speculation mechanism and the extensions necessary for the optimizations we propose.

Orca [8] and Shasta [112] were caching DSMs that provided prefetching in addition to caching to speedup data-parallel applications. However, they did not have explicit support for speculation like mechanisms for distributed locking and object versioning. [124, 96, 42, 30] explore the use of software transactional memory (STM) as a mechanism to achieve speculation. With STMs reading objects also locks them, which could limit the runtime parallelism. Moreover, STMs require frequent synchronization which is expensive and only increases in a distributed setting due to added communication and network latencies. In our approach, the use of synchronization is minimized by speculative lock-free reads and requires no synchronization until the data is ready to commit. Most recently, Dash et. al. presented a system that uses symbolic prefetching and caching to support distributed speculation using STM. For performance, they relax the STM to allow for lazy updates and invalidates to decrease inline communication latencies. However, we show that aggressive caching can hurt speculation and instead devise optimizations to decrease misspeculation cost and speedup misspeculated recomputations.

Many alternate approaches to distributed speculation have been explored. In the master-slave model of speculation, the master process runs approximate versions of the program, slaves validate correctness. The primary drawback is the ability to generate approximations for general purpose programs. [91] uses this same model in a distributed setting [136]. [62] uses home based release-consistency coupled with access predictors for future access to improve speculation performance. Our approach does not require release consistency by implementing the single-writer model. In [37], programmer has to express hints for parallelism, data dependence and data checking. Speculative execution is via independent processes; aborts simply terminate processes. OS page level monitoring is used to detect dependency violations. In contrast, our approach is more general purpose: we use a library based approach does not rely on low-level OS features. In addition, misspeculation results in retries and is therefore fully robust. D-BOP [110] attempts to speedup BOP [37] by predicting the outcome of a speculative execution based on the success/failure of the last speculation and avoiding un-promising executions all together. In our approach, we propose to eliminate the costly misspeculation checks whenever possible. [52] seeks to adapt the basic BOP in a distributed setup. Our approach is akin to the 'distributed validation, lazy update' (data is read only as needed; not broadcast), decreasing communication.

In [110] loads/stores uses the thread-id to enforce sequential consistency. Speculation is achieved via shadow copy; for the cluster, our approach assigns home nodes to variables; only tasks on home node update the variables, an approach similar to that taken by modern distributed computing frameworks like Hadoop which schedule compute close to storage. This strategy works particularly well for large data sets that can be easily partitioned among nodes in the cluster. Further, [110] proposes to replace bit-vectors for lock-synchronization with byte-arrays and using atomic primitives to read-write byte sized data as an alternate to lock-free data structures. However [110] also points the drawbacks and the lack of scalability of such an approach. In the distributed setup, our approach proposes piggybacking version numbers to decrease the communication cost associated with locks, which is the most expensive part of the distributed misspeculation check (aka distributed validation).

Systems like Hive use speculation in a distributed setting, but the motivation there is different: given the heterogeneity of hardware, multi-tenancy of the cluster and data replication, these systems schedule multiple instances of the same computation and pick up the results of the first one to complete and terminate other instances: they speculate that the occupancy and usage characteristics of the compute nodes could vary dynamically, so multiple instances are run hoping that it will be beneficial in the overall turn-around time, while we speculate that we can concurrently compute all values in parallel, which is a fundamentally different speculation.

Chapter 7

Conclusions and Future Work

7.1 Contributions

This dissertation contributes to enabling easy programming of *irregular applications* that crunch *large data* sets. It presents a shared-memory style programming system that is transformed into an object centric program that runs efficiently on a cluster of machines using software Distributed Shared Memory (DSM), thus simplifying the task of programming for the user. The programming interface comprises of very simple high-level extensions to the C++ language. The runtime employs sophisticated runtime techniques to speedup distributed software speculation and leverage the *InfiniMem* library to transparently spill data that cannot fit in local memory to the disk and transparently load the required data back into local node's memory. *InfiniMem* is built on a random-access efficient object data format on disk, well suited and tailored for the access patterns generated by irregular applications. Specifically, we highlight the contributions as follows:

Address Emerging Workloads

The focus of this thesis is irregular applications. Decreasing cost of storage and compute along with a massive proliferation of interest in data sciences for user data analysis has sparked a new interest in graph processing applications like PageRank. These applications process very large graphs as input. A typical computation usually requires a node and all its immediate neighbors – and this is iterated over all nodes in the graph. Parallelizing such loops with runtime cross iteration dependencies is a challenge and so is storing this data on disk when it cannot fit in available memory, since it generates random accesses. Our two-pronged solution of applying distributed software speculation and a random-access efficient data format on disk effectively tackles the emerging irregular applications.

Scale-up on Single Machines

While hardware cost has gone down and it is monetarily inexpensive to add machines to the cluster to handle larger data sizes, it is still wasteful if each machine is not utilized completely. The thesis presents *InfiniMem* as a way to scale-up individual machines by leveraging the local disk.

Given that the distribution of neighbors in the graph can be only known at runtime, there is no good one-size-fits-all solution to nicely organize the data structure for strided or similar 'regular' access when the inputs are large and need to be stored on disk. Therefore, computation on such inputs generally requires and generates random or 'irregular' accesses for data. Therefore, efficiently storing and retrieving such data from disk is a challenge which we solve here, not just for graphs, but for any arbitrary data type.

Additionally, we leverage multiple cores for balancing computation and communication on individual machines for efficient use of system resources while achieving significant performance gains.

Scale-out on Clusters

Addition of machines to the cluster theoretically scales-out the compute power. However, achieving significant gains practically is a different ball game. For example, if the application cannot be effectively parallelized, then compute power of the cluster does not matter to such programs. With irregular applications, since neighbors can only be discerned at runtime, the cross-iteration dependencies can also only be known at runtime, effectively requiring serial computation. We adapt software speculation DSM clusters, aiding with scale-out performance. In addition, we present techniques that leverage existing DSM cache structures to extract performance on the cluster, further pushing the scale-out performance.

Easy Programming and Fast Prototyping

Prior systems like CRL boasted rich features but required manual programming coupled with the need for arcane knowledge of the application domain, making it very hard to be generally adopted. Our language consists of just two simple to use language constructs. Our framework is also general-purpose, highly customizable and easy to program with. We also demonstrate that the *InfiniMem* framework can be used to quickly prototype specialized, single machine frameworks like GraphChi with relative ease.

7.2 Future Directions

Application in Diverse Domains

Domains like bioinformatics and computational genomics involve algorithms that process large graph-like connected structures. Therefore, such domains can also benefit from size oblivious irregular programming. Future work can explore the applicability and customization of our proposed solutions to those specific domains.

Additionally, since *InfiniMem* can handle arbitrary data types, studying additional domains could give us more insights into alternate data formats and how our proposed data format can be evolved, if needed.

Scale to Larger and non-DSM Clusters

Our experiments on distributed speculation were on a cluster with up to 8 machines, for a total of up to 256 cores. Scaling to larger clusters could potentially pose additional problems which do not manifest on smaller clusters. For example, bottlenecks from increased communication. We wish to explore this scalability in the future.

Distributed Memory (DM) systems, as opposed to Distributed Shared Memory (DSM) systems are in vogue today. Achieving speculation on DM systems will allow us to explore new problems and opportunities in the absence of caching, in message passing, work migration and related DM opportunities.

Reliable Computing

As *InfiniMem* spills data transparently to disk and transparently loads data from disk, *InfiniMem* can be applied in fault tolerant, reliable computing by way of automatic checkpointing and restore. Additionally, this format can also be used for persistent data when quick random access are a high priority.

Bibliography

[1] Kevork N Abazajian, Jennifer K Adelman-McCarthy, Marcel A Agüeros, Sahar S Allam, Carlos Allende Prieto, Deokkeun An, Kurt SJ Anderson, Scott F Anderson, James Annis, Neta A Bahcall, et al. The seventh data release of the sloan digital sky survey. *The Astrophysical Journal Supplement Series*, 182(2):543, 2009.

[2] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *ACM SIGARCH Computer Architecture News*, volume 16, pages 280–298. IEEE Computer Society Press, 1988.

[3] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Li-Jie Jin, Karthick Rajamani, and Willy Zwaenepoel. Adaptive protocols for software distributed shared memory. *Proceedings of the IEEE*, 87(3):467–475, 1999.

[4] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[5] Ching Avery. Giraph: large-scale graph processing infrastruction on hadoop. *Proceedings of Hadoop Summit. Santa Clara, USA:[sn]*, 2011.

[6] David A Bader and Kamesh Madduri. Gtgraph: A synthetic graph generator suite. *Atlanta, GA, February*, 2006.

[7] Henri E Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Rühl, and M Frans Kaashoek. Performance evaluation of the orca shared-object system. *ACM Transactions on Computer Systems (TOCS)*, 16(1):1–40, 1998.

[8] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18:190–205, 1992.

[9] James Bennett and Stan Lanning. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35, 2007.

[10] John K Bennett, John B Carter, and Willy Zwaenepoel. *Munin: Distributed shared memory based on type-specific memory coherence*, volume 25. ACM, 1990.

[11] Jon Berry and Greg Mackey. The multithreaded graph library, 2014. `https://software.sandia.gov/trac/mtgl`.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[13] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 247–258, 2008.

[14] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[15] 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.

[16] Hans P. Zima Bradford L. Chamberlain, David Callahan. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[17] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J Carey. A bloat-aware design for big data applications. In *Proceedings of the 2013 international symposium on International symposium on memory management*, pages 119–130. ACM, 2013.

[18] Protocol Buffers. Google's data interchange format, 2011.

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

[20] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. Asynchronous lease-based replication of software transactional memory. *Middleware 2010*, pages 376–396, 2010.

[21] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. Asynchronous lease-based replication of software transactional memory. *MIDDLEWARE 2010*, pages 376–396, 2010.

[22] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. In *ISCA*, pages 278–289, 2007.

[23] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[24] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010.

[25] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

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

[27] Thomas Cheatham, Amr Fahmy, Dan Stefanescu, and Leslie Valiant. Bulk synchronous parallel computing—a paradigm for transportable software. In *Tools and Environments for Parallel and Distributed Systems*, pages 61–76. Springer, 1996.

[28] Yi-Jen Chiang, Michael T Goodrich, Edward F Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149. Society for Industrial and Applied Mathematics, 1995.

[29] Alokika Dash and Brian Demsky. Automatically generating symbolic prefetches for distributed transactional memories. *Middleware 2010*, pages 355–375, 2010.

[30] Alokika Dash and Brian Demsky. Integrating caching and prefetching mechanisms in a distributed transactional memory. *Parallel and Distributed Systems, IEEE Transactions on*, 22(8):1284–1298, 2011.

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

[32] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with ordering guarantees. In *International Conference on Data Engineering (ICDE)*, pages 424–435, 2004.

[33] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[34] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[35] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Large-Scale Parallel Data Mining*, pages 245–260, 2000.

[36] Edsger W. Dijkstra. The origin of concurrent programming. chapter Cooperating sequential processes, pages 65–138. 2002.

[37] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior oriented parallelization. In *ACM SIGPLAN Notices*, volume 42, pages 223–234. ACM, 2007.

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

[39] Tarek El-Ghazawi and Lauren Smith. Upc: unified parallel c. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 27. ACM, 2006.

[40] Min Feng, Rajiv Gupta, and Yi Hu. Spicec: scalable parallelism via implicit copying and explicit commit. In *ACM SIGPLAN Notices*, volume 46, pages 69–80. ACM, 2011.

[41] Min Feng, Rajiv Gupta, and Iulian Neamtiu. Effective parallelization of loops in the presence of I/O operations. In *ACM Coference on Programming Language Design & Implementation (PLDI)*, pages 487–498, 2012.

[42] Joao Fernandes. Speculative execution on distributed and replicated software transactional memory systems. `http://www.gsd.inesc-id.pt/~ler/reports/joaofernandesea.pdf`.

[43] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, 2004.

[44] The MPI Forum. Mpi: Message passing interface. 1993. `http://www.mcs.anl.gov/research/projects/mpi/index.htm`.

[45] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[46] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[47] Douglas Gregor, Nick Edmonds, Alex Breuer, Peter Gottschling, Brian Barrett, and Andrew Lumsdaine. The parallel boost graph library. *The Trustees of Indiana University*, 2005.

[48] William Gropp. Mpich2: A new start for mpi implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–7. Springer, 2002.

[49] 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.

[50] Facebook Inc. Rocksdb. `http://rocksdb.org/`.

[51] Google Inc. Leveldb. `http://leveldb.org/`.

[52] Bryan Jacobs, Tongxin Bai, and Chen Ding. Distributed speculative program parallelization.

[53] C. Janna and M. Ferronato. 3d model of a steel flange, hexahedral finite elements., 2011. `http://www.cise.ufl.edu/research/sparse/matrices/Janna/Flan1565.html`.

[54] Yunlian Jiang and Xipeng Shen. Adaptive software speculation for enhancing the cost-efficiency of behavior-oriented parallelization. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 270–278. IEEE, 2008.

[55] Anoop Gupta John L. Henessey, Mark Heinrich. Cache-coherent distributed shared memory: perspectives on its development and future challenges. *Proceedings of the IEEE*, 87:418–429, 1999.

[56] Kirk Lauritz Johnson, M Frans Kaashoek, and Deborah A Wallach. *CRL: High-performance all-software distributed shared memory*, volume 29. ACM, 1995.

[57] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 252–263. ACM, 1997.

[58] E Jul, H Levy, N Hutchinson, and A Black. An object-oriented language and system that indirectly supports dsm through object mobility. *University of Washington Technical Report*, 1988.

[59] Magnus Karlsson and Per Stenström. Effectiveness of dynamic prefetching in multiple-writer distributed virtual shared-memory systems. *Journal of Parallel and Distributed Computing*, 43(2):79–93, 1997.

[60] George Karypis and Vipin Kumar. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[61] Ilya Katsov. Probabilistic data structures for web analytics and data mining. https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/, May 2012.

[62] Michael Kistler and Lorenzo Alvisi. Improving the performance of software distributed shared memory with speculation. *Parallel and Distributed Systems, IEEE Transactions on*, 16(9):885–896, 2005.

[63] Sai Charan Koduru, Min Feng, and Rajiv Gupta. Programming large dynamic data structures on a dsm cluster of multicores. In *7th International Conference on PGAS Programming Models*, page 126, 2013.

[64] Giorgos Kollias, Ananth Y Grama, and Zhiyuan Li. Asynchronous iterative algorithms. In *Encyclopedia of Parallel Computing*, pages 87–95. Springer, 2011.

[65] Masanori Koshiba, Shinji Maruyama, and E Hirayama. A vector finite element method with the high-order mixed-interpolation-type triangular elements for optical waveguiding problems. *Lightwave Technology, Journal of*, 12(3):495–502, 1994. `http://www.cise.ufl.edu/research/sparse/matrices/Dziekonski/dielFilterV3real.html`.

[66] Christos Kotselidis, Mohammad Ansari, Kimberly Jarvis, Mikel Lujan, Chris Kirkham, and Ian Watson. DiSTM: A software transactional memory framework for clusters. In *International Conference on Parallel Processing (ICPP)*, pages 51–58, 2008.

[67] Christos Kotselidis, Mohammad Ansari, Kimberly Jarvis, Mikel Lujan, Chris Kirkham, and Ian Watson. Investigating software transactional memory on clusters. In *IEEE Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–6, 2008.

[68] Elzbieta Krepska, Thilo Kielmann, Wan Fokkink, and Henri Bal. A high-level framework for distributed processing of large-scale graphs. In *Distributed Computing and Networking*, pages 155–166. Springer, 2011.

[69] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[70] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *ACM SIGPLAN Notices*, volume 44, pages 3–14. ACM, 2009.

[71] Vamsi K Kundeti, Sanguthevar Rajasekaran, Hieu Dinh, Matthew Vaughn, and Vishal Thapar. Efficient parallel and out of core algorithms for constructing large bi-directed de bruijn graphs. *BMC bioinformatics*, 11(1):560, 2010.

[72] Jérôme Kunegis. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*, pages 1343–1350, 2013. `http://userpages.uni-koblenz.de/~kunegis/paper/kunegis-koblenz-network-collection.pdf`.

[73] George Kurian, Jason E. Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C. Kimerling, and Anant Agarwal. Atac: a 1000-core cache-coherent processor with on-chip optical network. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 477–488, 2010.

[74] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[75] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[76] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO)*, 2004.

[77] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, 1(1):5, 2007. `http://snap.stanford.edu/data/amazon0302.html`.

[78] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[79] Yi Lin, Bettina Kemme, Marta Patiño Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Special Interest Group on Management of Data (SIGMOD)*, pages 419–430, 2005.

[80] Haiming Liu and Weiwu Hu. A comparison of two strategies of dynamic data prefetching in software dsm. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 6–pp. IEEE, 2001.

[81] Lixia Liu and Zhiyuan Li. Improving parallelism and locality with asynchronous algorithms. In *ACM Sigplan Notices*, volume 45, pages 213–222. ACM, 2010.

[82] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[83] Mary E. Mace. *Memory storage patterns in parallel processing.* Kluwer Academic Publishers, 1986.

[84] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz 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. ACM, 2010.

[85] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz 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. ACM, 2010.

[86] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 198–208, 2006.

[87] Rob Misek and Jon Purdy. Types of cache coherence. October 2006.

[88] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and Analysis of Online Social Networks. In *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.

[89] Luiz Rodolpho Monnerat and Ricardo Bianchini. Efficiently adapting to sharing patterns in software dsms. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 289–299. IEEE, 1998.

[90] Yasukazu Nakamura, Takehide Kosuge, Jun Mashima, Yuichi Kodama, Takatomo Fujisawa, Eli Kaminuma, Osamu Ogasawara, Kousaku Okubo, and Toshihisa Takagi. Ddbj-dna data bank of japan. 2014. `http://www.ddbj.nig.ac.jp`.

[91] Cosmin E Oancea, Jason WA Selby, Mark Giesbrecht, and Stephen M Watt. Distributed models of thread level speculation. In *PDPTA*, volume 5, pages 920–927, 2005.

[92] Victor Olman, Fenglou Mao, Hongwei Wu, and Ying Xu. Parallel clustering algorithm for large data sets with applications in bioinformatics. *IEEE Transactions on Computational Biology and Bioinformatics (TCBB)*, 6(2):344–352, 2009.

[93] MPI Open. Open source high performance computing, 2012.

[94] Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A Nayfeh, Monica S Lam, and Kunle Olukotun. *Software and hardware for exploiting speculative parallelism with a multiprocessor*. Computer Systems Laboratory, Stanford University, 1997.

[95] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[96] Roberto Palmieri, Francesco Quaglia, and Paolo Romano. Osare: Opportunistic speculation in actively replicated transactional systems. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 59–64. IEEE, 2011.

[97] David Patterson. The trouble with multi-core. *IEEE Spectrum*, 47(7):28–32, 2010.

[98] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Special Interest Group on Management of Data (SIGMOD)*, pages 165–178, 2009.

[99] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The tao of parallelism in algorithms. In *ACM SIGPLAN Notices*, volume 46, pages 12–25. ACM, 2011.

[100] Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. Safe programmable speculative parallelism. In *ACM Sigplan Notices*, volume 45, pages 50–61. ACM, 2010.

[101] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(1):3:1–3:55, 2011.

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

[103] Mukund Raghavachari and Anne Rogers. Understanding language support for irregular parallelism. In *Parallel Symbolic Languages and Systems*, pages 174–189. Springer, 1996.

[104] Lawrence Rauchwerger, Nancy M Amato, and David A Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th international conference on Supercomputing*, pages 137–146. ACM, 1995.

[105] Lawrence Rauchwerger and David A Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *Parallel and Distributed Systems, IEEE Transactions on*, 10(2):160–180, 1999.

[106] Lawrence Rauchwerger and David A. Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *Parallel and Distributed Systems, IEEE Transactions on*, 10(2):160–180, 1999.

[107] Kaushik Ravichandran and Santosh Pande. Multiverse: efficiently supporting distributed high-level speculation. *ACM SIGPLAN Notices*, 48(10):533–552, 2013.

[108] Paolo Romano, Roberto Palmieri, Francesco Quaglia, Nuno Carvalho, and Luís Rodrigues. Brief announcement: on speculative replication of transactional systems. In *Proc. of SPAA*, pages 69–71, 2010.

[109] Seyed H. Roosta. *Parallel Processing and Parallel Algorithms: Theory and Computation.* 2000.

[110] Peter Rundberg and Per Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism*, 3(1):2002, 2001.

[111] Barbara G. Ryder and Ben Wiedermann. Language design and analyzability: a retrospective. *Software: Practice and Experience (SP&E)*, 42(1):3–18, 2012.

[112] Daniel J Scales and Kourosh Gharachorloo. Shasta: A system for supporting fine-grain shared memory across clusters. In *PPSC*. Citeseer, 1997.

[113] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. In *Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 174–185, 1996.

[114] 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 *Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 297–306, 1994.

[115] Nir Shavit and Dan Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[116] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997.

[117] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146. ACM, 2013.

[118] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[119] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[120] Jeremy Siek, L Lee, and Andrew Lumsdaine. The boost graph library (bgl). http://www.boost.org, 2000.

[121] Guy L Steele Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 218–231. ACM, 1989.

[122] Lubos Takac and Michal Zabovsky. Data analysis in public social networks. In *Int'l. Scientific Conf. & Int'l Workshop Present Day Trends of Innovations*, 2012.

[123] Chunqiang Tang, DeQing Chen, Sandhya Dwarkadas, and Michael L. Scott. Integrating remote invocation and distributed shared state. In *IEEE Parallel & Distributed Processing Symposium (IPDPS)*, 2004.

[124] Cristian Ţăpuş and Jason Hickey. Speculations: providing fault-tolerance and recoverability in distributed environments. In *Proceedings of the Second conference on Hot topics in system dependability*, pages 10–10. USENIX Association, 2006.

[125] TM Team et al. Apache mahout project, 2014.

[126] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[127] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms and Visualization*, 50:161–179, 1999.

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

[129] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. Aspire: exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 861–878. ACM, 2014.

[130] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *International Conference on Data Engineering (ICDE)*, pages 422–433, 2005.

[131] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[132] Mitsuo Yokokawa, Fumiyoshi Shoji, Atsuya Uno, Motoyoshi Kurokawa, and Tadashi Watanabe. The K computer: Japanese next-generation supercomputer development project. In *ISLPED*, pages 371–372, 2011.

[133] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[134] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

[135] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. Flashgraph: processing billion-node graphs on an array of commodity ssds. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 45–58. USENIX Association, 2015.

[136] Craig Zilles and Gurindar Sohi. Master/slave speculative parallelization. In *Microarchitecture, 2002.(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 85–96. IEEE, 2002.