Towards a Systematic Study of Big Data Performance and Benchmarking

Thesis · September 2016

DOI: 10.13140/RG.2.2.28960.38406

1 author:

Saliya Ekanayake
Indiana University Bloomington
23 PUBLICATIONS **108** CITATIONS

SEE PROFILE

TOWARDS A SYSTEMATIC STUDY OF BIG DATA PERFORMANCE AND BENCHMARKING

Saliya Ekanayake

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics and Computing
Indiana University
October 2016

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Geoffrey Fox, Ph.D.

Andrew Lumsdaine, Ph.D.

Judy Qiu, Ph.D.

Haixu Tang, Ph.D.

September 28, 2016

To my wife Kalani, son Neth, and our parents.

Acknowledgements

This dissertation represents work done over many years. I have been fortunate to be among many who cared for me and guided me throughout this endeavor. I am humbled by their support, without which none of this would have been possible.

I express my sincere gratitude to my mentor and adviser Prof. Geoffrey Fox for his invaluable research insight and constant encouragement. His remarkable expertise in parallel programming has helped me to solve numerous challenges throughout this research. I am honored to have conducted this Ph.D under his guidance.

I am indebted to my committee members Prof. Judy Qiu, Prof. Andrew Lumsdaine, and Prof. Haixu Tang for their equally invaluable guidance and support. I am privileged to have taken their courses and to have worked with them on this research. Their wisdom has helped to shape me into a researcher and scientist. I also thank my former committee members Prof. Kent Dybvig and Prof. Arun Chauhan for their feedback and guidance that made it possible to reach the early Ph.D. milestones.

Dr. Sanjiva Weerawarana paved the way for the start of my graduate studies at Indiana University. I thank him for being an exceptional mentor to me and many other Sri Lankan students. My deepest gratitude also goes to Vishaka Nanayakkara and Dr. Sanath Jayasena of University Moratuwa, Sri Lanka for their invaluable support and guidance.

It has been a delight to work among my brilliant peers at the Digital Science Center (DSC) research group. Supun Kamburugamuve has coauthored several papers with me and supported this research in many ways. I have enjoyed working with Jerome Mitchell and treasured his friendship during this work. Pulasthi Wickramasinghe has greatly supported with testing applications and authoring papers. I also extend my thanks to Andrew Young, TakLon Wu, and BingJing Zang.

Former colleagues – all doctors now – have helped me immensely during the early years of this research. Jaliya Ekanayake, my brother, constantly encouraged and guided me during this Ph.D. He has been a role model throughout my life and has always cared for me. Thilina Gunarathne has assisted countless times with his technical expertise and guidance. I also thank Ruan Yang, and Jong Choi

My sincere thanks also go to Gary Miksik, Mary Nell Shiflet, Allan Streib, Adam Hughes, Scott Beason, Julie Overfield, and the staff of the School of Informatics and Computing for providing continual help during my Ph.D. study.

Suresh Marru of the Science Gateways Group has guided me in many ways both professionally and as a friend. Eric Holk is an incredible programmer and a colleague whose LaTeXsupport made it possible to write this dissertation as swiftly as possible. Prof. Dan Friedman's course has been a complete eye-opener and changed my view on programming

languages. I have also taken great pleasure in attending Prof. Amr Sabry's and Dr. Kent Dybvig's courses.

I would like to thank John McCurley for his untiring proofreading of this dissertation.

I have been blessed to be surrounded by friends whose love and care made our time in Bloomington unforgettably wonderful. I thank Milinda Pathirage, Isuru Suriyaarachchi, Supun Kamburugamuve, Udayanga Wickramasinghe, Amila Jayasekara, and the rest of my Sri Lankan friends.

Words cannot explain the love, support, and encouragement of my parents. It has been a ritual for my mother, Padmini Ekanayake, to start her day by calling me throughout this journey. My father, Prof. Punchibanda Ekanayake, has never failed to inspire me through his courage and dedication. They have taught me to be humble yet strong in the face of whatever the life has to offer.

None of this would have been possible if not for the one special person, Kalani, who has been at my side through tears and joys. Her devoted love and courage as my wife are the pillars of support that I stand on today. Finally, my dear son, Neth, is too little to understand what daddy is writing here, but his smiles have wiped away my pains and have given me reason to be strong more than ever!

Saliya Ekanayake

TOWARDS A SYSTEMATIC STUDY OF BIG DATA PERFORMANCE AND BENCHMARKING

Big data queries are increasing in complexity and the performance of data analytics is of growing importance. To this end, Big Data on high-performance computing (HPC) infrastructure is becoming a pathway to high-performance data analytics. The state of performance studies on this convergence between Big Data and HPC, however, is limited and ad hoc. A systematic performance study is thus timely and forms the core of this research.

This thesis investigates the challenges involved in developing Big Data applications with significant computations and strict latency guarantees on multicore HPC clusters. Three key areas it considers are thread models, affinity, and communication mechanisms. Thread models discuss the challenges of exploiting intra-node parallelism on modern multicore chips, while affinity looks at data locality and Non-Uniform Memory Access (NUMA) effects. Communication mechanisms investigate the difficulties of Big Data communications. For example, parallel machine learning depends on collective communications, unlike classic scientific simulations, which mostly use neighbor communications. Minimizing this cost while scaling out to higher parallelisms requires non-trivial optimizations, especially when using high-level languages such as Java or Scala. The investigation also includes a discussion on performance implications of different programming models such as dataflow

and message passing used in Big Data analytics. The optimizations identified in this research are incorporated in developing the Scalable Parallel Interoperable Data Analytics Library (SPIDAL) in Java, which includes a collection of multidimensional scaling and clustering algorithms optimized to run on HPC clusters.

Besides presenting performance optimizations, this thesis explores a novel scheme for characterizing Big Data benchmarks. Fundamentally, a benchmark evaluates a certain performance-related aspect of a given system. For example, HPC benchmarks such as LINPACK and NAS Parallel Benchmark (NPB) evaluate the floating-point operations (flops) per second through a computational workload. The challenge with Big Data workloads is the diversity of their applications, which makes it impossible to classify them along a single dimension. Convergence Diamonds (CDs) is a multifaceted scheme that identifies four dimensions of Big Data workloads. These dimensions are: problem architecture, execution, data source and style, and processing view.

The performance optimizations together with the richness of CDs provide a systematic guide to developing high-performance Big Data benchmarks, specifically targeting data analytics on large, multicore HPC clusters.

Geoffrey Fox, Ph.D.

Andrew Lumsdaine, Ph.D.

Judy Qiu, Ph.D.

Haixu Tang, Ph.D.

Contents

List of Figures

CHAPTER 1

Introduction

The concept of Big Data has evolved from its early hype to a norm within a short period of time. Google pioneered Big Data processing with the advent of the MapReduce [**22**] framework for analyzing its large collection of web data. The MapReduce implementation gained wide publicity for reasons such as the ease of programming, built-in fault tolerance, and horizontal scalability over commodity clusters. Google's implementation was proprietary, and with the growing interest in MapReduce, Yahoo developed an open-source version and released it under Apache Software Foundation (ASF) as Apache Hadoop [**95**]. It has since been adopted heavily in both academia and industry. Currently, an abundance of open-source Big Data frameworks and libraries exist in the Apache Big Data Stack (ABDS) [**36**], including popular frameworks like Apache Spark, Flink, and Storm that support complex batch and streaming data analytics beyond MapReduce. It is worth noting here that most of these frameworks and applications are based on high-level languages such as Java, Scala, and Python, which is uncommon in classic high-performance computing (HPC) applications.

Big Data queries are becoming increasingly complex, and there is a rising trend to bridge Big Data and HPC, especially for applications such as parallel machine learning that exhibit intensive computations and communication. This means performance is of utmost importance. To this end, there are over a dozen Big Data benchmarks; however, while they

are designed to compare similar products, they do not provide insight into performance scaling within a node or across nodes. Further, their coverage is limited and ad hoc.

This thesis explores the challenges of developing high-performance, parallel machine-learning applications on HPC environments. A primary challenge with parallel machine learning is its sensitivity to performance variations in individual tasks. To elaborate, these algorithms are typically iterative in nature and require collective communications that are not easily overlapped with computations; hence, the performance is susceptible to communication overheads and noise caused by slow-performing tasks. Beyond the nature of these applications, the use of high-level languages such as Java, as mentioned previously, on multicore Non-Uniform Memory Access (NUMA) nodes brings out additional challenges in keeping constant performance when scaling over the many cores within a node as well as across nodes. This research identifies three key factors that significantly affect performance. These factors are thread models, affinity strategies for threads and processes, and communication mechanisms. Moreover, we discuss several other factors of importance when developing high-performance applications using Java and Object Oriented Programming (OOP). We present optimization techniques to bring performance closer to traditional HPC applications written in languages like C.

To study performance we carefully look at three major frameworks: Message Passing Interface (MPI), Apache Spark [**98**], and Apache Flink [**1**, **12**]. Two parallel machine learning algorithms – K-Means clustering and Multidimensional Scaling (MDS) – are used to evaluate these frameworks using an Intel Haswell HPC cluster consisting of both 24-core and 36-core nodes. The experiments focus on four aspects: scaling over the many cores of a node,

efficiently exploiting the maximum parallelism of a node, scaling across nodes, and scaling over varying data sizes. These tests form the basis of our systematic study of Big Data performance.

The rest of the thesis is organized as follows. Chapter 2 provides some background information on different parallel programming models used to write Big Data applications. Chapter 3 presents a survey of existing Big Data benchmarks, technical improvements, and benchmark design guidelines. Chapter 4 introduces Scalable Parallel Interoperable Data Analytics Library (SPIDAL), a parallel machine-learning library that is contributed to as part of this research. It also presents a novel, multifaceted Big Data classification scheme to overcome the limitations of existing classification approaches. Chapter 5 identifies the major factors affecting Java parallel machine-learning and presents optimizations to improve performance to near native levels. Chapter 6 presents a detailed performance study of two machine-learning applications highlighting the effects of the optimizations discussed in the previous chapter. Finally, Chapter 7 summarizes and concludes the work of this research.

CHAPTER 2

Background

Before diving into the details of performance and benchmarking it is helpful to understand the different programming models that write parallel Big Data applications. The following sections describe Shared Memory (SM), Distributed Memory (DM), hybrid SM and DM, and Dataflow (DF) programming models, all of which are popular among both Big Data and scientific communities.

2.1. Shared Memory (SM)

The SM model, as the name implies, presents a shared memory layer to the program. Developers consider an SM program as a collection of tasks running in a single address space as shown in Figure 2.1.

The language supporting a shared memory model is responsible for mapping it onto the physical computing system. Typically, tasks are mapped to threads and the implementation

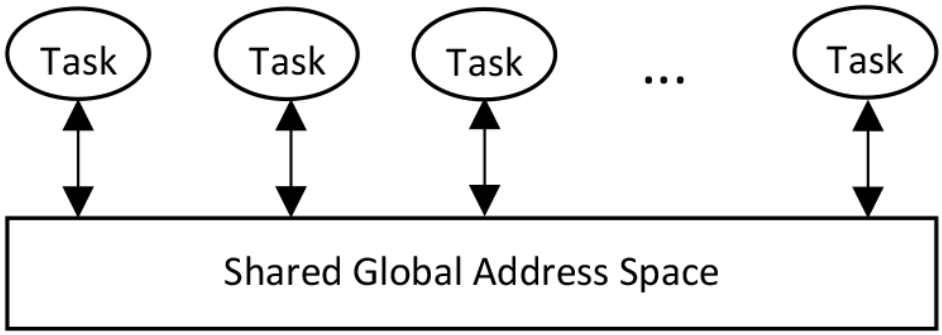


FIGURE 2.1. Programmer's view of the SM model

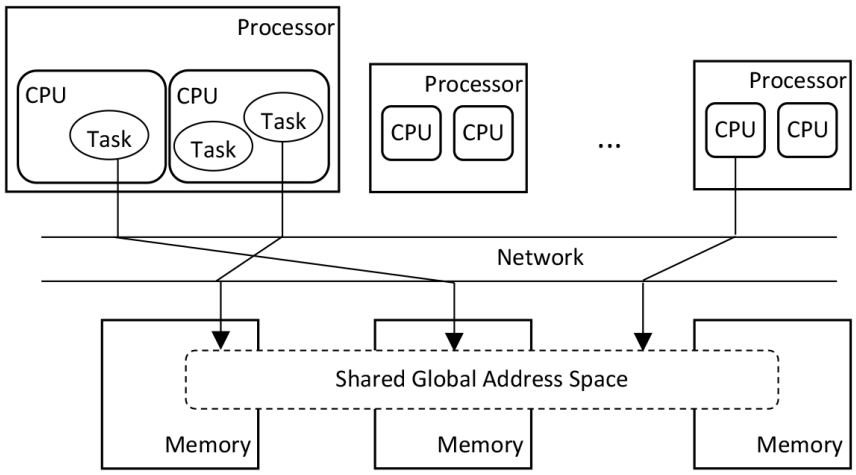


FIGURE 2.2. SM model implementation on distributed machines

may determine if the mapping is one-to-one or many-to-one. Although the common use of the SM model is local to a node, it is possible to implement the SM model over distributed machines **[76]**. This necessitates communications over the network as shown in the illustration in Figure 2.2. Note that the leftmost processor is enlarged to show the mapping of tasks onto central processing units (CPUs), and the network layer is placed between the processor and memory of each machine for clarity, but this does not imply uniform memory access. Although not shown in the figure, tasks can have unshared local address spaces in physical memory local to the processor running a particular task. The advantage of this model to a developer is the ability to share data across tasks, unconstrained by the notion of ownership, which requires explicit communication to share data. The caveat, however, is the need to synchronize data access to preserve the expected behavior of the program.

The most common implementation of the SM model is the thread support found in almost all programming languages. Portable Operating System Interface (POSIX) threads (or 'Pthreads') is a thread specification for Linux Operating Systems (OSs) and is supported in the C language. Other languages such as Java and Python have their own thread implementations. There are also high-level SM libraries such as Open Multi-Processing (OpenMP), Intel Thread Building Blocks (TBBs) [**81**], Microsoft's Task Parallel Library (TPL) [**64**], and Habanero Java (HJ) [**50**] to name a few. We briefly introduce OpenMP and HJ below as they were used in performance evaluations in Section 6.

2.1.1. OpenMP. OpenMP is one of the most popular implementations of the SM model. It provides an Application Programming Interface (API) based on compiler directives to express parallelism explicitly along with a set of library routines and environment variables. OpenMP implementations support C, C++, and Fortran languages and are portable enough to run on both Linux- and Windows-based platforms. Before the introduction of parallel tasks in version 3.0, OpenMP programs by default followed a fork-join execution model, where a master thread would fork worker threads when necessary. Once the parallel work was done, the threads would exit, leaving only the master thread again. The number of threads spawned in a parallel region can be determined by the programmer by setting either the `OMP_NUM_THREADS` environment variable or calling the `omp_set_num_threads(int num_threads)` method before the start of a parallel region. The threads are numbered from zero to one minus the total number of threads, making it possible for different threads to take different execution paths. OpenMP also provides several synchronization constructs to the programmer to avoid race conditions

when multiple threads work on shared data, such as critical, atomic, barrier, master, ordered, and flush. Also, OpenMP version 3.0 introduced tasks in which a thread may generate any number of other tasks to be run by a pool of threads. The moment of execution of a task is decided by the runtime and completion of tasks and can be enforced using task synchronization.

2.1.2. HJ. HJ [50] is a Java-based thread library similar to OpenMP. It provides two parallel constructs to do loop parallelism: `forall` and `forasync`. The former includes an implicit synchronization at the end of the parallel loop. These two also have `forallChunked` and `forasyncChunked` versions to partition the data into chunks, so an asynchronous task may handle a block of data rather than creating tasks per item of data. Besides loop-parallel constructs, HJ provides other kinds of asynchronous task creation similar to OpenMP and coordination constructs, such as `isolated`, futures, phasers, and actors.

2.2. Distributed Memory (DM)

The DM model presents a segmented address space in which each segment is local only to a single task. A programmer's view of a DM program is as a collection of individual tasks acting on their own local address spaces as shown in Figure 2.3.

The DM model obligates the programmer to partition data structures that need to be distributed among tasks manually and requires explicit communication between tasks for synchronization. Communication is shown in dashed lines in Figure 2.3. Typically, DM encourages Single Program Multiple Data (SPMD) programming, which works well for many parallel algorithms. The best-known implementation of DM is MPI, discussed in

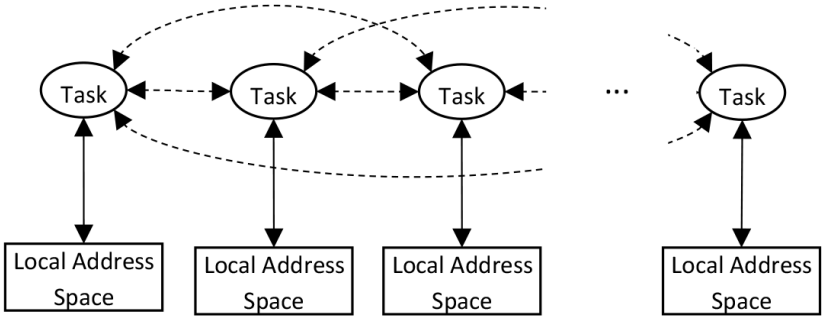


FIGURE 2.3. Programmer's view of the DM model

Section 2.2.1. Big Data frameworks such as Apache Hadoop [**95**], Twister [**26**], and Harp [**99**] are also realizations of the DM model but restrict themselves to the MapReduce [**22**] model and its extensions. One advantage of following such a restrictive DM model is the ability to provide fault tolerance at the framework level, which is especially important for programs running on commodity clusters.

2.2.1. MPI. MPI is an API specification allowing inter-process communication via message passing. Over the years it has become the *de facto* standard in realizing DM parallel programming. The most frequently used implementations of MPI are OpenMPI [**43**], MPICH [**44**], MVAPICH [**48**], and vendor-specific implementations, such as Intel MPI [**52**], Cray MPI [**51**], and Microsoft MPI [**16**]. While these commonly support C, C++, and Fortran, other bindings exist for languages such as Java, C#, Python, Ruby, and Perl.

The execution model of an MPI program requires the user to specify the total number of processes. In the MPI-1 specification the number of processes is a constant throughout the execution of the program. MPI-2 relaxes this constraint to support dynamic process

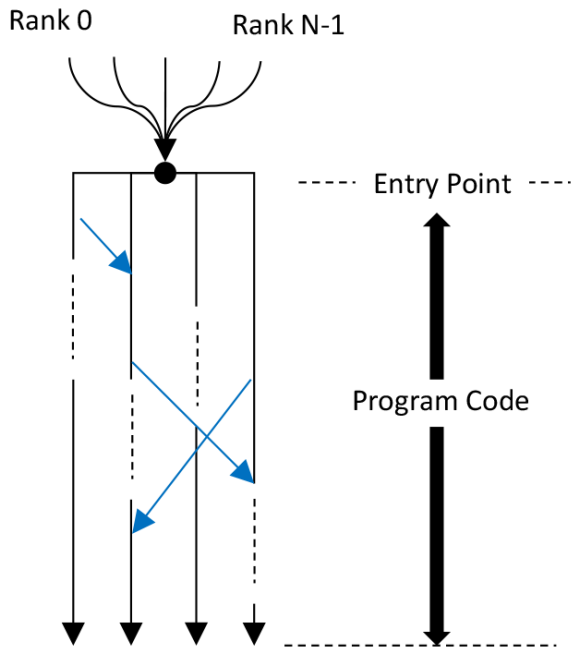


FIGURE 2.4. Execution model of an MPI program

creation and connection of previously existing processes. All processes belong to the `MPI_COMM_WORLD` communicator and are ranked from 0 to N-1, where N is the total number of processes. Figure 2.4 illustrates the general execution model of an MPI program.

The usual practice of MPI programs is to execute the same application by all the processes resembling the SPMD model. The flow of execution, however, may differ across processes depending on their rank. The dashed line segments in Figure 2.4 denote this difference in control flow. These processes operate on separate data entities and data sharing is made explicit through communication between processes as identified by blue arrows in the figure.

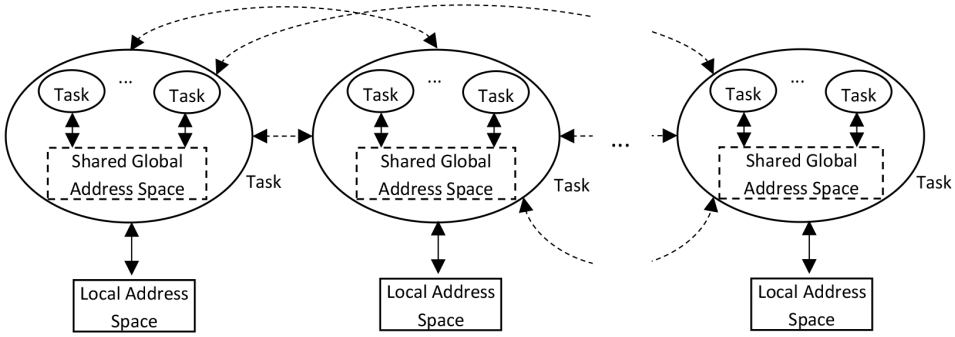


FIGURE 2.5. Programmer's view of the hybrid model

MPI is a complete programming model capable of implementing any parallel algorithm. Also, MPI implementations have been perfected for high-performance, especially in HPC environments with tightly coupled nodes.

2.3. Hybrid of SM and DM

It is common to combine the SM and DM models to implement parallel programs. A programmer's view is shown in Figure 2.5. There are two types of tasks visible in the figure: inter-node tasks and intra-node tasks. These are represented in large and small ellipses, respectively. The classic realization of this model is to combine OpenMP with MPI. The outer tasks, the processes, share their address space with threads running inside of them. While it is not restricted to initiating communications at thread-level, in practice communications happen at the process level while threads perform compute-only tasks. In multicore environments, threads are usually preferred over processes for intra-node tasks, as they incur less communication cost than processes do. However, there are other performance implications for using threads, which we will discuss in a later section.

2.4. Dataflow Programming

Dataflow programming models parallel computation as a series of data-transformation steps. Apache Spark, Apache Flink, and Apache Storm are popular dataflow frameworks that emerged with Big Data. Overall, dataflow is similar in style to the classic HPC workflow systems except at a finer level. For example, in workflow systems data transformations happen across coarse-grained components, such as separate applications. These components appear as black boxes to the workflow system, which essentially coordinates them by passing data as inputs and outputs. In a dataflow system the transformations are tasks that are visible to the framework and are usually composed of transformation constructs exposed by the framework. Generally, the following operations are supported in dataflow systems.

(1) **Map** - Represents a transformation function that is applied to data partitions in parallel.

(2) **Filter** - Similar to the Map but represents a logic to filter out data elements.

(3) **Project** - Selects part of the data elements.

(4) **Group** - Groups data items based on a provided key attribute.

(5) **Reduce** - Reductions, as the name implies, are used to combine and reduce an input dataset. For example, the plus operator may be used to sum a collection of numbers.

(6) **Aggregate** - Similar to reductions, aggregations reduce data elements. These are essentially built-in reduce functions.

(7) **Join** - Joins two datasets based on a key attribute.

(8) **Cross** - Produces the cross product of two datasets.

(9) **Union** - Produces the union of two datasets.

The dataflow model offers a more declarative approach to parallel programming than other DM models and is being adopted in the industry to process Big Data. However, the suitability of this model for parallel analytics requiring complex computations and communications is still being investigated. In our performance study presented in Section 6, we observed that current dataflow systems could benefit from classic HPC concepts, especially when used within HPC environments.

CHAPTER 3

Related Work

As the diversity of Big Data continues to increase, a vast collection of benchmarks have emerged to evaluate the performance of Big Data systems. While the majority of the benchmarks have originated from Big Data companies to evaluate performance of their systems, other research, such as BigDataBench [**93**], is aimed at providing a repository of Big Data benchmarks and guidelines on how to select a few representative benchmarks to evaluate a system. Beyond establishing these benchmarks the next line of research relating to performance concerns is the technical improvements of Big Data frameworks and libraries. Further research explores approaches toward consensus around the benchmarking of Big Data systems. The following sections present current work on these three related areas, starting with the current landscape of benchmarks.

3.1. Survey of the Current Benchmarks

There is a range of Big Data benchmarks in the current literature, and the following sections describe a selected few from different areas. To highlight the diversity of Big Data benchmarks, we start with two of the prominent HPC benchmarks, LINPACK (LINPACK) and NAS Parallel Benchmarks (NPBs), which are primarily used to evaluate computation performance of HPC systems.

3.1.1. LINPACK and Its Variants. The lineage of LINPACK **[23]** includes the following three benchmark suites.

- LAPACK (LAPACK) **[4]**: the shared memory implementation
- ScaLAPACK (ScaLAPACK) **[9]**: the parallel distributed memory implementation
- HPL (HPL) [1]: Top500's [2] yardstick

These are kernel solvers for dense linear systems of the form $Ax = b$. The strategy is to use lower upper (LU) factorization followed by a solver that totals $2n^3/3 + 2n^2$ floating-point operations (flops). The performance metric is flops per second, generally mega or giga flops per second (Mflop/s of Gflop/s).

The LINPACK benchmark report **[24]** includes results from three benchmarks: LINPACK Fortran n=100, LINPACK n=1000, and HPL. The first is a sequential, Fortran-based solver for a matrix of order 100. The rules specify that no change other than compiler optimizations are allowed for this case. The second benchmark is for a matrix of order 1000 with relaxed rules such that the user can replace the LU factorization and solver steps. The report also includes results exploiting shared-memory parallelism in a fork-join style for the n=1000 test. The HPL benchmark relaxes both the choice of implementation and problem size. Its parallel algorithm for distributed memory systems is explained in the HPL algorithm report [3], and a scalable implementation is packaged into the HPL software distribution that scales both with the amount of computation and communication volume as long as the memory usage per processor is maintained **[25]**.

[1] http://www.netlib.org/benchmark/hpl
[2] http://www.top500.org/
[3] http://www.netlib.org/benchmark/hpl/algorithm.html

3.1.2. NAS Parallel Benchmarks (NPBs). NPBs are a set of kernel and pseudo applications derived from Computational Fluid Dynamics (CFD) applications. They are meant to compare the performance of parallel computers and to serve as a standard indicator of performance [**67**]. The original NPB 1, which is a paper-and-pencil specification, includes five kernels and three pseudo applications. Optimized MPI parallel implementations habe been available since version 2.3.

The original benchmark set was extended with multi-zone (MZ) implementations of the original block tridiagonal (BT), scalar pentadiagonal (SP), and LU pseudo applications. MZ versions are intended to exploit multiple levels of parallelism, and the implementations use MPI plus threading with OpenMP. [4] NPB was further extended to include benchmarks that evaluate unstructured computation, parallel I/O, and data movement. Alongside NPB, GridNPB, another set of benchmarks, was introduced in order to rate the performance of grid environments.

A notable feature in NPB is its well-defined benchmark classes: small (S), workstation (W), standard, and large. The standard class is further divided into subclasses A, B, and C with problem size increasing roughly four times from going one class to the next. The large class also introduces D, E, and F subclasses where the problem size increase is roughly 16 times. A detailed description of the actual problem sizes for each class is available on the NPB website. [5] We capture this property in our proposed classification strategy as well.

3.1.3. BigDataBench. BigDataBench [**66**, **94**] is a benchmark suite targeting Internet services. There is a total of 34 implemented benchmarks (or 'workloads' following the

[4] http://openmp.org/wp/
[5] http://www.nas.nasa.gov/publications/npb_problem_sizes.html

authors), which fall into any of five application domains: search engines, social networks, e-commerce, multimedia data analytics, and bioinformatics. Moreover, some of these benchmarks have multiple implementations to suite different Big Data frameworks. The implementations use several components of the ABDS [6] and some of their commercial adaptations. An extracted summary of benchmarks is given in Table 3.1.

Version 3.1 of the BigDataBench handbook mentions that each workload is quantified over 45 micro-architectural-level metrics in the following categories: instruction mix, cache behavior, TLB behavior, branch execution, pipeline behavior, offcore request, snoop response, parallelism, and operation intensity. The original paper **[94]** also presents three user-perceivable metrics: process requests per second (RPS), operations per second (OPS), and data processes per second (DPS). Note that each of these is relevant only for some workloads.

BigDataBench presents two things: implications of data volume and benchmark characterizations. The paper **[94]** presents the importance of testing with increasing loads to determine the performance trends in each case. The metrics, million instructions per second (MIPS) and cache misses per 1000 instructions (MPKI) are given to elaborate this fact. The benchmark characterization measures operation intensity and effects of hierarchical memory. In conclusion they present that the kind of benchmarks tested in BigDataBench show relatively low ratios of computation to memory accesses compared to traditional HPC benchmarks. Further, they show that L3 caches show the lowest MPKI numbers for these

[6] http://hpc-abds.org/kaleidoscope/

benchmarks and that a possible cause of seeing higher MPKI values in lower-level caches (L1, L2) could be due to the use of deep software stacks.

TABLE 3.1. Benchmark summary of BigDataBench.

Application Domain	**Operation or Algorithm**	**Software Stack**
Search Engine	Sort, Grep, WordCount, Index, PageRank, Nutch Server, Read, Write, Scan	Hadoop, Spark, MPI, Nutch, HBase, MySQL
Social network	K-means, Connected Components (CC), BFS	Hadoop, Spark, MPI
E-commerce	Select, Aggregate, and Join queries, Collaborative Filtering (CF), Nave Bayes, Project, Filter, Cross Product, OrderBy, Union, Difference, Aggregation	Impala, Hive, Shark
Multimedia	BasicMPEG, SIFT, DBN, Speech Recognition, Image Segmentation, Face Detection	MPI
Bioinformatics	SAND, BLAST	Work Queue, MPI

In addition to providing a large number of benchmarks and metrics, BigDataBench also presents a way to reduce the number of benchmarks that one would require in order to assess a system comprehensively. Instead of characterizing a benchmark by 45 dimensions (micro-architectural metrics), the strategy involves picking the most uncorrelated dimensions

with the help of running Principal Component Analysis (PCA) [**94**] and then clustering the benchmark performance vectors with K-means clustering to form groups of similar benchmarks. A representative benchmark is then picked from each cluster either by picking one close to the edge of a cluster or the middle of a cluster. There are two lists of such shortlisted benchmarks presented in [**66**].

3.1.4. HiBench. HiBench [**47**] is a Hadoop benchmark suite intended to evaluate MapReduce-style applications. It captures the interest of the Big Data community to use Hadoop and its ecosystem Pig, Hive, Mahout, etc. to areas such as machine learning, bioinformatics, and financial analysis. The introduction of HiBench, as it authors claim, is to overcome the limited representation and diversity of existing benchmarks for Hadoop at this time. The benchmarks they compare are the following sort programs: GridMix [7], DFSIO [8], and Hive [9]. A few reasons why these benchmarks do not produce a fair evaluation are: 1) they do not exhibit computations compared to real applications; 2) they do not feature data access outside of map tasks; and 3) they represent only analytical database queries (Hive benchmarks), which do not evaluate MapReduce over a broad spectrum of Big Data analysis.

HiBench introduces micro-benchmarks and real-world applications. These micro-benchmarks include the original Sort, WordCount, and TeraSort from Hadoop. The applications are Nutch indexing, PageRank, Bayesian classification, K-means clustering, and EnhancedDFSIO. The EnhancedDFSIO could be identified as a micro-benchmark in today's

[7] https://developer.yahoo.com/blogs/hadoop/gridmix3-emulating-production-workload-apache-hadoop-450.html
[8] http://epaulson.github.io/HadoopInternals/benchmarks.html#dfsio
[9] https://issues.apache.org/jira/browse/HIVE-396

context and extends the original DFSIO to include measure-aggregated I/O bandwidth. HiBench evaluates these benchmarks for job running time and throughput, aggregated Hadoop Distributed File System (HDFS) bandwidth, utilization of CPU, memory and I/O, and data access patterns, i.e. data sizes in map-in, map-out/combiner-in, combiner-out/shuffle-in, and reduce-out stages. In conclusion the authors claim HiBench represents a wider range of data analytic problems with diverse data access and resource utilization patterns. The latest release of HiBench is version 3.0, completed on October 2014.

3.1.5. Graph500. Graph500 [10], unlike other Big Data benchmarks, is intended to evaluate a variety of architectures, programming models, and languages and frameworks against data intensive workloads. It brings to light the point that systems targeted for traditional physics simulations may not be the best for data intensive problems. The benchmark performs a breadth-first graph search and defines six problem classes denoted as levels 10 through 15. These indicate the storage in bytes required to store the edge list such that for a given level L the size will be in the order of 10^L.

There are two timed kernels in Graph500. Kernel 1 creates a graph representation from an edge list and Kernel 2 performs the Breadth-First Search (BFS). Kernel 2 is run multiple times (usually 64), each with a different starting vertex. After each run a soft validation is run on results. The soft validation checks for properties of a correct BFS tree rather than verifying if the resultant BFS tree is the one for the input graph and the particular starting vertex. The performance metric of Graph500 defines a new rate, traversed edges per

[10] http://www.graph500.org/

second (TEPS). It is defined as TEPS $= m/time_{k2}$, where m is the number of edges including any multiple edges and self-loops, and $time_{k2}$ is Kernel 2's execution time.

3.1.6. BigBench. BigBench [**41**,**42**] is an industry-led effort to defining a comprehensive Big Data benchmark. It emerged with a proposal that appeared in the first Workshop on Big Data Benchmarking (WBDB) [**79**]. It is a paper-and-pencil specification, but it comes with a reference implementation to get started. BigBench models a retailer and benchmarks 30 queries around it, covering five business categories depicted in the McKinsey report [**68**].

The retailer data model in BigBench addresses the three V's volume, variety, and velocity of Big Data systems It covers variety of data by introducing structured, semi-structured, and unstructured data in the model. While the first is an adaptation from the TPC-DS [11] benchmark's data model, the semi-structured data represents the click stream on the site, and unstructured data represents product reviews submitted by users. Volume and velocity are covered with a scale factor in the specification that determines the size for all data types, and a periodic refresh process based on TPC-DS's data maintenance, respectively.

Part of the BigBench research is on data generation, which includes an extension to the popular Parallel Data Generation Framework (PDGF) [**78**] that generates the click-stream (semi-structured) data and a novel synthetic reviews (unstructured text data) generator, TextGen, which is seamlessly integrated with PDGF.

There are a total of 30 queries covering 10 classes from five business categories. While these cover the business side well, they also cover 3 technical dimensions data source,

[11] http://www.tpc.org/information/benchmarks.asp

processing type, and analytic technique. Data source coverage is to represent all three structured, semi-structured, and unstructured data in the queries. Given that BigBench is a paper-and-pencil specification, the queries are specified using plain English. While some of these could be implemented efficiently with Structured Query Language (SQL) or HiveQL [12] like declarative syntaxes, the others could benefit from a procedural implementation like MapReduce or a mix of these two approaches. The processing-type dimension assures that the queries manage reasonable coverage of these three types. BigBench identifies three analytic techniques for answering queries: statistical analysis, data mining, and simple reporting. The paper does not define a performance metric for future work, but it suggests taking a geometric-mean approach such that $\sqrt[30]{\prod_{i=1}^{30} P_i}$ where P_i denotes execution time for query i. It also presents their experience implementing and running this query end-to-end on Teradata Aster Database Management System (DBMS).

In summary, BigBench is in active development at present and provides good coverage for business-related queries over a synthetic data set. Plans are set for a Transaction Processing Council (TPC) proposal with its 2.0 version as well.

3.1.7. MineBench. MineBench **[70]** is a benchmark targeted for data-mining workloads and presents 15 applications covering five categories as shown in Table 3.2.

TABLE 3.2. Minebench applications

Application	Category	Description
		Continued on next page

[12] https://cwiki.apache.org/confluence/display/Hive/LanguageManual

Application	Category	Description
ScalParC	Classification	Decision tree classification
Nave Bayesian	Classification	Simple statistical classifier
SNP	Classification	Hill-climbing search method for DNA dependency extraction
Research	Classification	RNA sequence search using stochastic Context-Free Grammars
SVM-RFE	Classification	Gene expression classifier using recursive feature elimination
K-means	Clustering	Mean-based data partitioning method
Fuzzy K-means	Clustering	Fuzzy logic-based data partitioning method
HOP	Clustering	Density-based grouping method
BIRCH	Clustering	Hierarchical Clustering method
Eclat	Association Rule Mining	Vertical database, Lattice transversal techniques used
Apriori	Association Rule Mining	Horizontal database, level-wise mining based on Apriori property

Continued on next page

Application	Category	Description
Utility	Association Rule Mining	Utility-based association rule mining
GeneNet	Structure Learning	Learning Gene relationship extraction using microarray-based method
SEMPHY	Structure Learning	Learning Gene sequencing using phylogenetic tree-based method
PLSA	Optimization	DNA sequence alignment using Smith-Waterman optimization method

It has been a while since MineBench's latest release in 2010, but it serves as a good reference for the kind of applications used in data mining. Moreover, these are real-world applications and the authors provide OpenMP-based parallel versions for most of them. The input data used in these applications come from real and synthetic data sets of small, medium, and large size.

A performance characterization of data-mining applications using MineBench is studied in two papers [70, 75]. The architectural characterization study [75] is of particular interest for a couple of reasons. First, it justifies the need to introduce a new benchmarking system by identifying the diversity of data-mining applications. It does so by representing each application as a vector of its performance counters and using K-means clustering to group

them. While applications from other benchmarks such as SPEC INT, SPEC FP, MediaBench, and TPC-H tend to cluster together, data-mining applications fall under multiple clusters. Second, it characterizes the applications based on: 1) execution time and scalability, 2) memory hierarchy behavior, and 3) instruction efficiency. While it is expected from any benchmark to have a study of performance and scalability, we find the other two dimensions are equally important and adaptable towards studying Big Data benchmarks as well.

3.1.8. LinkBench. LinkBench [**5**] is a benchmark developed at Facebook to evaluate its graph-serving capabilities. Note that LinkBench evaluates a transactional workload, which is different from a graph-processing benchmark like Graph500 that runs an analytic workload. LinkBench is intended to serve as a synthetic benchmark to predict the performance of a database system serving Facebook's production data, thereby reducing the need to perform costly and time-consuming evaluations that mirror real data and requests.

The data model of LinkBench is a social graph where nodes and edges are represented using appropriate structures in the underlying datastore, for example using tables with MySQL. The authors have studied in detail the characteristics of Facebook's data when coming up with a data generator that would closely resemble it.

The workload is also modeled after careful study of actual social transactions. The authors consider several factors such as access patterns and distributions, access patterns by data type, graph structure and access patterns, and update characterization in coming up with an operation mix for the benchmark.

The design includes a driver program that generates data and fires up requester threads with the operation mix. The connections to the data store are handled through LinkBench's

graph-store implementation, which currently includes support for MySQL back-ends. Most of the information for the benchmark is fed through a simple configuration file, which makes it easy to adapt for different settings in the future.

Primary metrics included in the benchmark are operation latency and mean operation throughput. The other metrics include price/performance, CPU usage, I/O count per second, I/O rate MB/s, resident memory size, and persistent storage size.

3.1.9. BG Benchmark. BG [**2**] emulates read and write actions performed on a social networking datastore and benchmarks them against a given service-level agreement (SLA). These actions originate from interactive social actions like view profile, list friends, view friend requests, etc. BG defines a data model and lists the social actions it benchmarks in detail in its paper [**2**]. It introduces two metrics to characterize a given datastore, as given below.

- **Social Action Rating (SoAR)**: The highest number of completed actions per second agreeing to a given SLA.
- **Socialites**: The highest number of simultaneous threads that issue requests against the datastore and satisfy the given SLA.

An SLA requires that for some fixed duration: 1) a fixed percentage of requests observing latencies equal or less than a given threshold, and 2) the amount of unpredictable data is less than a given threshold. Quantifying unpredictable data is an offline process done through log analysis at the granularity of a social action.

BG implementation consists of three components: a BG coordinator, a BG client, and a BG visualization deck. There can be multiple clients, and they are responsible for data

and action generation. The coordinator communicates with clients to instruct them how to generate data and emulate actions based on the given SLA. It also aggregates the results from clients and make them available to the visualization deck for presentation.

3.1.10. SparkBench: A Comprehensive Benchmarking Suite for In-Memory Data Analytic Platform Spark. SparkBench is a suite of benchmarks for evaluating the performance of an Apache Spark cluster. Table 3.3 summarizes the workloads in SparkBench.

SparkBench comes with a data generator that can produce datasets with varying sizes. The reported metrics are job execution time, input data size, and data process rate. A few metrics under development are shuffle data, RDD size, and resource consumption. Also, integration with monitoring tools is in development.

TABLE 3.3. SparkBench workloads

Application	Category	Input Dataset
Logistic Regression	Machine Learning	Wikipedia
Support Vector Machine	Machine Learning	Wikipedia
Matrix Factorization	Machine Learning	Amazon Movie Reviews
Page Rank	Graph Computation	Google Web Graph
SVD++	Graph Computation	Amazon Movie Reviews
Triangle Count	Graph Computation	Amazon Movie Reviews
Hive	SQL Engine	E-commerce
RDD Relation	SQL Engine	E-commerce

Continued on next page

Application	Category	Input Dataset
Twitter	Streaming Application	Twitter
Page Review	Streaming Application	Page View Data Gen

Table 3.4 describes the datasets used in SparkBench.

TABLE 3.4. SparkBench Datasets

Data Sets	Description
Wikipedia	6938018 articles.
Google Web Graph	875713 nodes, 5105039 edges.
Amazon Movie Reviews	7911684 reviews, 889176 movies, 253059 users.
E-commerce Transactions	38275 orders, 8 columns. 240332 items, 7 columns.

SparkBench is intended for the following purposes.

- Quantitative comparison of different platforms and cluster setups
- Quantitative comparison of different Spark configurations and optimizations
- To provide guidance when deploying a Spark cluster

3.1.11. TPCx-HS. TPCx-HS stands for "Transaction Processing Council Express"^{tm} Hadoop Sort. It is the industry's first standard for benchmarking Big Data systems and models continuous system availability of 24 hours a day, 7 days a week. The core of TPCx-HS is built around the TeraSort [**34**] implementation that comes with Apache Hadoop. The benchmark consists of two five-step runs. The steps are as follows.

(1) **HSGen**: Generates input data that must be replicated 3-ways and written on a durable medium.

(2) **HSDataCheck**: Verifies cardinality, size, and replication of generated data.

(3) **HSSort**: Samples the input data and sorts data. The sorted output must be replicated 3-ways and written on a durable medium.

(4) **HSDataCheck**: Verifies cardinality, size, and replication of the sorted output.

(5) **HSValidate**: Validates correctness of the sorted output.

The two runs are identified as the **performance run** and the **repeatability run** based on their TPCx-HS performance metric, $HSph@SF$, defined as $HSph@SF = SF/(T/3600)$. In this definition, SF identifies the scale factor of data, which can be 1TB, 3TB, 10TB, 30TB, 100TB, 300TB, 10000TB, 30000TB, or 100000TB. The run with the lower metric is denoted as the performance run and the other as the repeatability run. Besides $HSph@SF$, TPC-xHS defines two other metrics: price-performance and availability date. Also, there is an energy metric that define power per performance. Details of these metrics can be found in the TPC-xHS specification [**20**].

3.1.12. YCSB. Yahoo Cloud Serving Benchmark (YCSB) [**15**] is a benchmark suite developed by Yahoo to evaluate cloud serving systems such as HBase, Cassandra, Infinispan, and MongoDB. TPC-C benchmark has been the norm for evaluating traditional database systems. However, cloud serving systems do not necessarily support the Atomicity, Consistency, Isolation, Durability (ACID) properties of traditional databases. Also, it is hard to capture their usability through a single scenario, as in TPC-C. YCSB presents an extensible

workload generator and an extensible interface to support a variety of cloud serving systems. The primary intention of YCSB is to provide a tool to perform an apples-to-apples comparison between such systems. YCSB propose two benchmark tiers: performance and scaling. For performance it evaluates the latency of the system for queries. For scaling it tests two things. The first is to evaluate the latency as the number of servers and database size increase. The second is to test the elasticity of the system by loading a constant dataset and dynamically adding more servers. Details of the workloads and performance results are given in the YCSB paper **[15]**.

3.1.13. Berkeley Big Data Benchmark. Traditional databases serve queries, typically in SQL. Data warehousing as a concept emerged with Big Data as analytics were performed over traditional databases. Berkeley Big Data Benchmark is designed to evaluate such data warehousing frameworks such as Amazon Redshift, Apache Hive, SparkSQL, Cloudera Impala, and Stringer. The benchmark evaluates these frameworks against a handful of relational queries across different data sizes. More information about the queries and the initial results can be found in Berkeley Big Data Benchmark **[62]**

3.1.14. CloudSuite. CloudSuite **[33]** is benchmark suite studying the behavior of scale-out workloads using performance counters. It evaluates five scale-out workloads as listed below.

- **Data Serving**: Evaluates Cassandra database using YCSB generated data.
- **MapReduce**: Performs Bayesian classification using Mahout library in Hadoop for Wikipedia pages.
- **Media Streaming**: Benchmarks Darwin Streaming Server.

- **SAT Solver**: Evaluates Klee SAT Solver, which is compute intensive and is usually designed to run in HPC environments.
- **Web Frontend**: Benchmarks a web-based social event calendar, Olio.
- **Web Search**: Evaluates an index serving node of the distributed version of Nutch with an index size of 2GB.

In addition to these scale-out workloads, the authors have also tested five traditional benchmarks to understand their low-level performance counters. These benchmarks are PARSEC 2.1, SPEC CINT 2006, SPECweb09, TPC-C, TPC-E, and Web Backend. Note that the Web Backend benchmark tests a standard MySQL database engine.

The findings of these workloads suggest there is a great mismatch between scale-out workloads and the optimization found in standard server hardware. A detailed analysis is given in the CloudSuite paper [**33**].

3.2. Technical Improvements

Apache Hadoop [**95**] is the first open-source software product that emerged with Big Data. It implements Google's MapReduce [**22**] model and has been well recognized both in industry and academia. Over the last decade it has gone through major changes to improve functionality as well as performance. Decoupling Hadoop's resource management from its MapReduce engine in version 2.0 was the biggest improvement. This allowed Hadoop to be extended easily for purposes beyond MapReduce. Also, the previous version had a single point of failure, as it could only run one `Namenode`, which is responsible for managing metadata in HDFS. This is resolved in version 2.0 and above by allowing multiple `Namenode` daemons. Also, the newer version can scale up to 10,000 nodes per

cluster, whereas the earlier could reach a maximum of 4,000 nodes. The Yet Another Resource Negotiator (YARN) paper discusses these improvements in detail [**91**].

Twister [**26**] is an iterative MapReduce framework developed at Indiana University during the same time when Hadoop was released. While it is possible to develop iterative applications using Hadoop as a chain of MapReduce jobs, the frequent data reading and writing between iterations significantly degrades the application's performance. Twister improved on this aspect with built-in iterative support and in-memory data transfers. Performance of Twister has shown orders of magnitude of improvement over Hadoop. Harp [**100**] is another project at Indiana University developing a collective communication framework over Hadoop to quicken machine learning applications. It brings the best of classic HPC collectives to the Hadoop domain.

Similar to the improvements made in Twister, Apache Spark [**98**] improved upon in-memory data transfer to overcome the performance bottleneck with Hadoop. It also introduced a different programming model than MapReduce with its Resilient Distributed Datasets (RDDs). The dataflow it introduced quickly caught the attention of the Big Data community. In 2014 Spark developers initiated project Tungsten [**96**] to improve Spark's performance. It provided three performance improvements: memory management, cache-aware computation, and code generation. Memory management is about explicitly managing memory to avoid Java Virtual Machine (JVM) overheads. Cache-aware computation is used to exploit cache hierarchy efficiently. Code generation concerns the binary code generation used for some Spark queries.

The high-performance streaming data paper [55] discuss several performance improvements on Apache Storm. One of the main contributions of this paper is the introduction of efficient collective algorithm implementations to Storm. It also applies shared-memory-based communication to tasks running within the same node. Recently, Twitter Inc. introduced a complete revamp of Storm under the name Apache Heron [**61**]. The primary goal of Heron has been to overcome performance challenges with Storm. Heron's blog post [**80**] describes the details of these improvements.

In addition to the improvements in Big Data frameworks, a significant amount of research has been devoted to improving computation and communication performance, especially for Java programs. To this end, several libraries are available in Java supporting the MPI model in HPC environments. OpenMPI provides by far the best Java support for MPI [**73**]. Guillermo et al. [**90**] discuss the performance of some of the other Java MPI frameworks such as MPJ-Express [**7**] and JaMP [**59**] within HPC environments and introduce FasMPJ [**32**] as a high-performance, purely Java-based MPI implementation. We have found the OpenMPI's Java support to outperform FastMPJ due to its use of native C implementation of the actual communications.

Rajesh et al. [**58**] discusses actor-based frameworks to exploit the multicore machines, and Flink uses this actor model for handling concurrency. Besides this there are other published works [**11**, **14**, **77**] on improving performance in NUMA multicore environments. Garbage Collection (Garbage Collection (GC)) also plays a vital role in high performance and Maria et al. [**13**] shows how to optimize the Java GC in multi-core NUMA machines.

3.3. Benchmarking Guidelines

With the rise of Big Data applications comes the challenge of defining guidelines to develop standard benchmarks. While research is still ongoing, Baru et al. [**8**] have carried out most of the initial work toward standardizing benchmark design. They have identified five design concerns as described below.

3.3.1. Component vs. End-to-End Benchmark. Component benchmarks measure the performance of a single or a few components of a system, whereas end-to-end benchmarks assess an entire system. The advantage of the component benchmarks is that they are relatively easier to specify and run. Also, if the components expose standard APIs then the benchmark could be run as-is, for example using a benchmark kit. Standard Performance Evaluation Corporation (SPEC) CPU benchmark **[19]** and TeraSort are examples of component benchmarks. TPC benchmarks are examples of end-to-end types where an entire system is modeled. The complexity of large systems, however, could hinder the specification and development of standardized benchmark kits.

3.3.2. Big Data Applications. This topic is about the coverage of Big Data benchmarks, as there are various use cases that fall under Big Data. For example, the large volumes of data generated in the Large Haron Collider (LHC) or the retail systems of businesses such as Amazon, Ebay, and Walmart all deal with Big Data. With such diversity it is difficult to find a single benchmark to cover them all. Further, it is an open question as to whether Big Data benchmarks should model concrete applications or should model abstractions based on real applications. Modeling a concrete application is favorable in that it is possible to

use real-world examples for the benchmark process. Two real applications the paper [8] suggest to model are a retailer system as modeled in TPC benchmarks and a social website like Facebook or Netflix.

3.3.3. Data Sources. Data for Big Data benchmarks could either be synthetic or real. While it is preferred to model benchmarks around real use cases, using real datasets is not preferable. Real data needs to be downloaded and stored, which is impractical due to the large size. Also, real data would reflect only certain properties, which makes benchmarking inaccurate. Synthetic data on the other hand is relatively easy to generate using parallel data generators and can be designed to incorporate all necessary characteristics.

3.3.4. Scaling. The topic scaling is about being able to extrapolate results based on the performance obtained with the System Under Test (SUT). Allowing scalable system sizes in the benchmark could lead to vendors increasing system size to get better performance numbers than their competitors. This could also impose a huge financial burden on smaller vendors. Therefore, it is preferable to be able to extrapolate results. Also, it is important to consider the notion of 'elasticity' that is usually associated with cloud systems. It is the ability to add resources dynamically depending on load. Moreover, system failures could result in dynamically removing resources. Considering these two cases, the ability to extrapolate results becomes more relevant with Big Data benchmarks.

3.3.5. Metrics. Besides reporting performance numbers, it is important to include cost-based metrics similar to the $(price/performance)$ found in the TPC benchmarks. It is also

important to consider energy costs as well as setup costs, especially with Big Data, as large infrastructure may be a significant factor in the overall cost.

CHAPTER 4

Scalable Parallel Interoperable Data Analytics Library (SPIDAL)

SPIDAL is a suite of machine-learning algorithms optimized for Big Data analytics on large HPC clusters. Primarily, it implements several multidimensional scaling and clustering algorithms. The following sections discuss these algorithms, their use cases, and other tools in SPIDAL. We also introduce a novel scheme for systematically classifying Scalable Parallel Interoperable Data Analytics Library (SPIDAL) and other Big Data benchmarks at the end.

4.1. DA-MDS

Deterministic Annealing Multidimensional Scaling (DA-MDS) implements an efficient weighted version of Scaling by MAjorization of a COmplicated Function (SMACOF) **[10]** that effectively runs in $O(N^2)$ compared to the original $O(N^3)$ implementation **[84]**. It also uses a deterministic annealing optimization technique **[60, 82]** to find the global optimum instead of local optima. Given an *NxN* distance matrix for *N* high-dimensional data items, DA-MDS finds *N* lower-dimensional points (usually three for visualization purposes) such that the sum of error squared is the minimum. The error is defined as the difference between mapped and original distances for a given pair of points. DA-MDS also supports arbitrary

weights and fixed points —data points that already have the same low-dimensional mapping. Currently, the optimized implementation is based on Java MPI and threads. Two other versions based on Spark and Flink are under development.

4.2. DA-PWC

Deterministic Annealing Pairwise Clustering (DA-PWC) also uses the concept of DA, but for clustering [39, 82]. Its time complexity is $O(NlogN)$, which is better than existing $O(N^2)$ implementations [37]. Similar to DA-MDS it accepts an NxN pairwise distance matrix and produces a mapping from point number to cluster number. It can also find cluster centers based on the smallest mean distance, i.e. the point with the smallest mean distance to all other points in a given cluster. If provided with a coordinate mapping for each point, it can produce centers based on the smallest mean Euclidean distance and Euclidean center. Its implementation is based on Java MPI and threads.

4.3. DA-VS

Deterministic Annealing Vector Sponge (DA-VS) is a recent addition to the SPIDAL library based on MPI and Java threads. It can perform clustering in both vector and metric spaces. Algorithmic details and an application of DA-VS to protein data is available in the study by Fox et al. paper [35].

4.4. MDSasChisq

Multidimensional Scaling as Chi-squared (MDSasChisq) is a general MDS implementation based on the LevenbergMarquardt algorithm [65]. Similar to DA-MDS it supports

arbitrary weights and fixed points. Additionally, it supports scaling and rotation of MDS mappings, which is useful when visually comparing 3D MDS outputs for the same data but with different distance measures and other runtime parameters. The implementation is Java MPI and thread based.

4.5. K-Means Clustering

K-Means is a well-known clustering algorithm, which typically operates on vectors. Scalable Parallel Interoperable Data Analytics Library (SPIDAL) recently included an optimized Java MPI and thread-based implementation suitable for high-performance computing (HPC) clusters. It also provides a native C implementation based on MPI and OpenMP.

4.6. Elkan's K-Means Clustering

Elkan's algorithm [**31**] is an improvement over the classic K-Means clustering. It utilizes the triangle inequality to reduce the number of distance computations in K-Means. An efficient parallel implementation of this algorithm is present in SPIDAL as an option in DA-VS.

4.7. WebPlotViz

WebPlotViz [**56**] is a web-based point visualization tool in the SPIDAL suite. In a typical analysis carried out using the Scalable Parallel Interoperable Data Analytics Library (SPIDAL) suite, the output consists of a plot file that represents the input items as three-dimensional points. The coordinates of these points are such that the distance between a given pair of points is equal to the dissimilarity between the corresponding data items.

The dissimilarity between input data points can be calculated any number of ways. For example, the distance between two sequences in gene sequence data is calculated as the percent identity of their Smith-Waterman-Gotoh (SWG) [87]. Besides displaying points, WebPlotViz also supports graph structures, making it possible to visualize structures such as phylogenetic trees and other complex networks. Moreover, it supports streaming data, where a series of plots can be animated as a time series. This is relevant when analyzing frequently changing data such as stock market values. The highlight of WebPlotViz is that it provides a novel approach for biologists and other scientists to inspect the structure of input data visually.

4.8. SPIDAL Use Cases

4.8.1. Gene Sequence Clustering. One of the strongest use cases of the Scalable Parallel Interoperable Data Analytics Library (SPIDAL) suite is the analysis of gene sequences [**27**, **46**, **83**, **85**, **89**]. The objective is to classify sequences into groups of similar characteristics. The initial step of this process is to compute dissimilarity between each pair of sequences using an alignment algorithm such as Smith-Waterman-Gotoh (SWG) [87] or Needleman-Wunsch (NW) [**71**]. In SPIDAL, we have implemented parallel algorithms to perform this alignment for large sequence datasets. The dissimilarities are presented as an *NxN* binary matrix, where *N* is the number of sequences. We feed this matrix to Deterministic Annealing Multidimensional Scaling (DA-MDS) and Deterministic Annealing Pairwise Clustering (DA-PWC) to produce a three-dimensional mapping of sequences and clustering information. These two results are combined and visualized in WebPlotViz. The pipeline described here is illustrated in Figure 4.1. Note that in Figure 4.1 DA-PWC (P3) has an

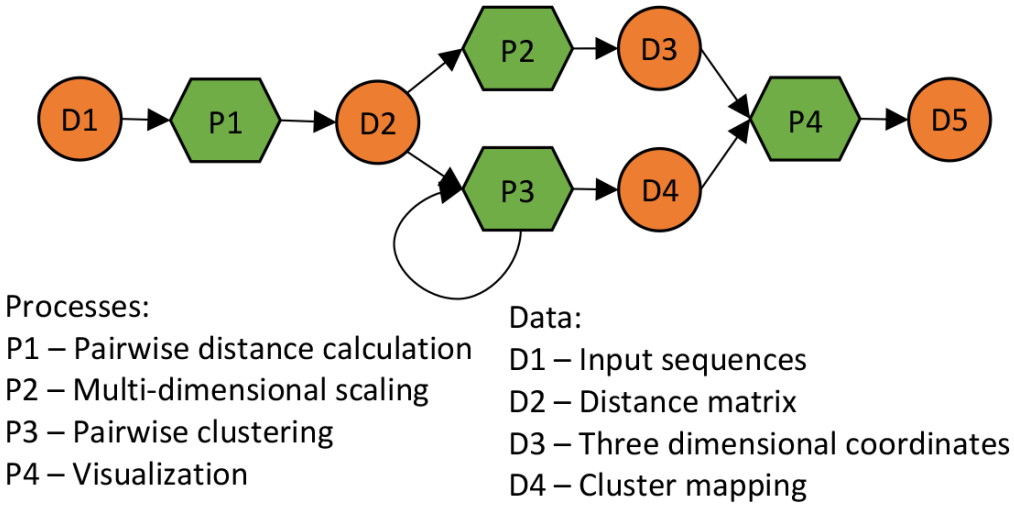
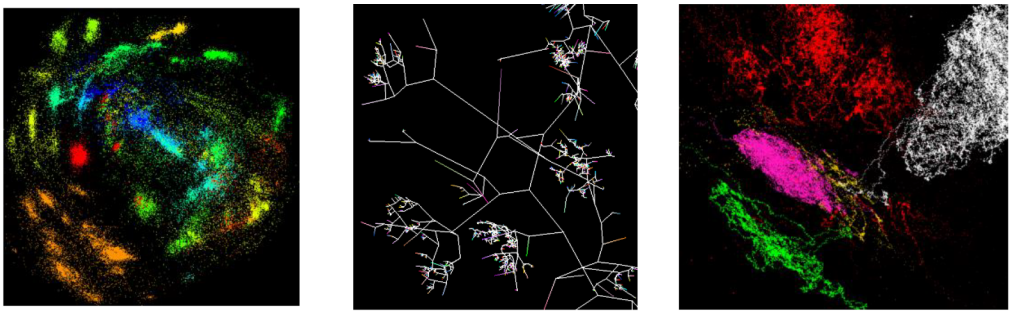


FIGURE 4.1. Gene sequence analysis pipeline

arrow going to itself indicating that clustering is a multistage process. Generally, it is better to cluster the full dataset into a smaller number of clusters and then do further work on those clusters separately.

Figure 4.2 shows a few results of analyzing gene sequences. The leftmost image shows the clusters for a 100,000-fungi-sequence dataset. The middle image illustrates how the results from DA-MDS can be used to create a three-dimensional phylogenetic tree and display using WebPlotViz. The rightmost picture shows another dataset where input is not sequences but a set of vectors.

4.8.2. Stock Market Analysis. In a recent study we used SPIDAL algorithms to analyze the behavior of stocks [**57**]. The data for this experiment contained around 7,000 distinct stocks with daily values available at 2,750 distinct times. These data are from Jan 01, 2004 to Dec 31, 2015. We used information such as Stock ID, Date, Symbol, Factor to Adjust Volume,



(a) 100,000 fungi sequences (b) 3D phylogenetic tree (c) 3D plot of vector data

FIGURE 4.2. Gene sequence analysis snapshots

Factor to Adjust Price, Price, and Outstanding Stocks to represent each stock as a vector. The analysis process is illustrated in Figure 4.3.

Figure 4.4 and Figure 4.5 show the relative change in stocks using one-day values. The filled circles show the final values, and the zigzag paths show their change in value over the time period. The time-series-data-visualisation capabilities of WebPlotViz made it possible to see these changes in a way similar to video playback.

4.9. Convergence Diamonds: A Novel Approach to Benchmark Classification

The number of Big Data benchmarks published within the last few years is impressive given the relatively young age of Big Data compared to Big Simulations, its well-established counterpart in the scientific community. The SPIDAL algorithms discussed above also can be used to define a set of Big Data benchmarks to go along with those discussed in Section 3.1. While having a collection of benchmarks is helpful in addressing the diversity of Big Data applications, a systematic classification is necessary to identifying their similarity and coverage. BigDataBench **[66]**, for example, provides a mechanism to pick a subset of

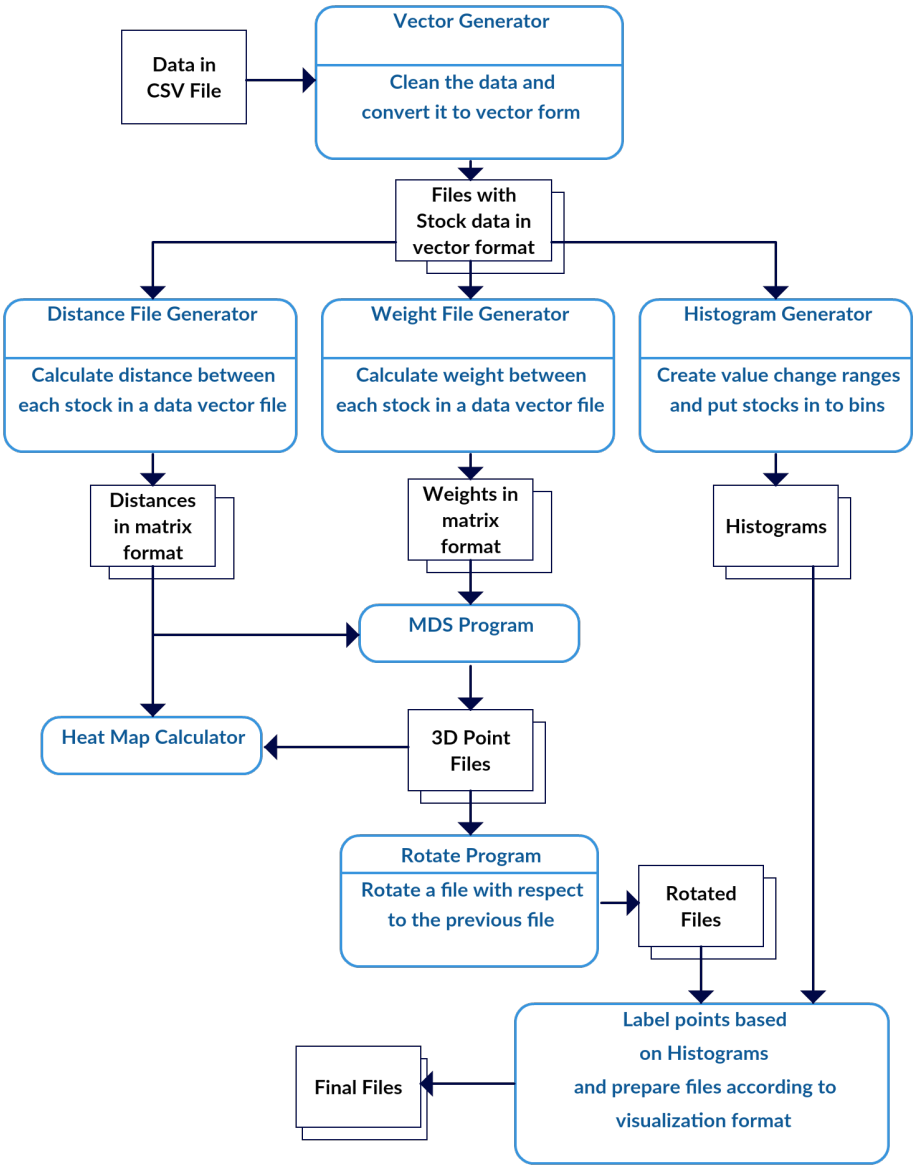


FIGURE 4.3. Stock data analysis process

its benchmarks in evaluating a Big Data system [**53**] based on the micro-architectural-level `perf` metrics available in Linux operating systems. The idea is to represent each benchmark as a vector of different runtime metrics, such as load and store, cache misses, Translation Lookaside Buffer (TLB) misses, and off-core requests. A total of 45 metrics spanning across

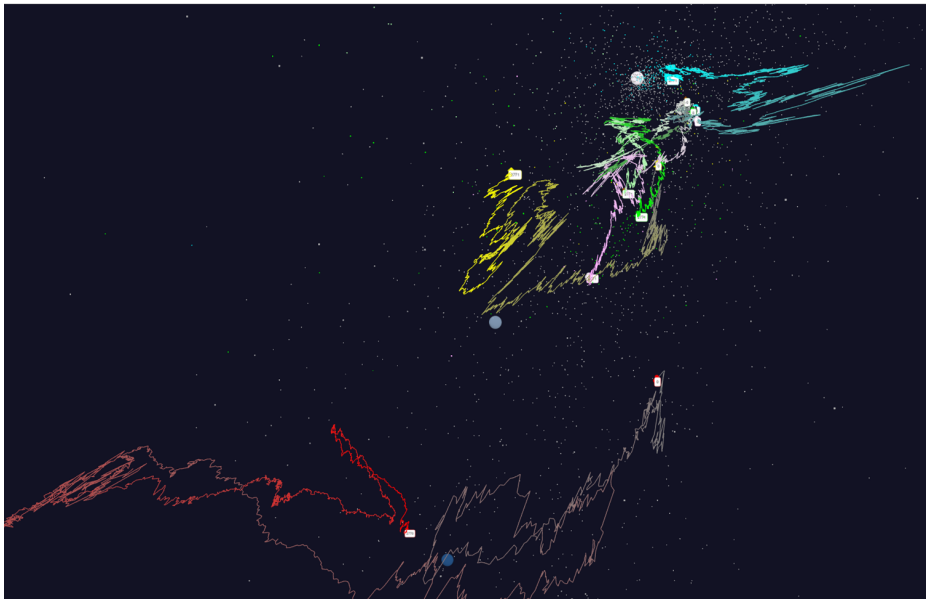


FIGURE 4.4. Relative changes in stocks using one day values

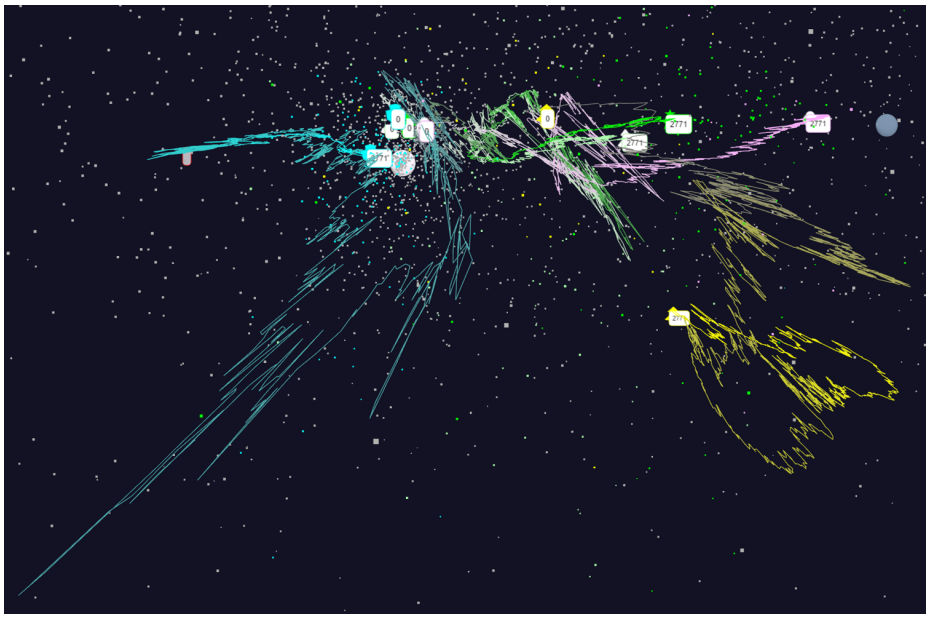


FIGURE 4.5. Relative changes in stocks using one day values expanded

nine categories is used in BigDataBench. The vectors are then processed through PCA to remove correlated metrics and clustered using K-means algorithm. A representative benchmark from each cluster is selected to produce the final subset of benchmarks. Also,

a hierarchical clustering algorithm is run on the PCA output to produce a dendrogram showing the similarity between benchmarks.

While runtime characteristics are an important measure in studying benchmarks, the downside is that one needs first to run and collect those metrics before being able to compare a new benchmark against existing ones. Also, such schemes do not capture the high-level features of the benchmark. Berkeley Dwarfs [**6**] presents a pattern-based approach to benchmark classification. The Dwarfs classification, however, is intended for HPC simulation applications and therefore is limited in applicability to covering the diverse range of Big Data applications. This section introduces Convergence Diamonds (CDs), which is a novel approach to investigating and classifying the properties of both Big Data and Big Simulations, thereby solving the limitations of the existing approaches. CDs is an extension over our previous Ogre classification [**40**].

CDs present a multidimensional and multifaceted classification. The dimensions and facets are illustrated in Figure 4.6

The four dimensions of CDs are:

- **Problem Architecture View (PAV)**: Describes the overall structure of the problem.
- **Execution View (EV)**: Lists facets related to the execution of an application, such as performance metrics.
- **Data Source and Style View (DSSV)**: Describes data of the application. This is more relevant to Big Data applications than to scientific applications.
- **Processing View (PV)**: Facets of the application model separated for Big Data and Big Simulations.

Of the 64 features illustrated in Figure 4.6, the following describes a select few for each dimension.

4.9.1. Facets in Problem Architecture View.

- **Pleasingly Parallel (PAV-1)**: Indicates the processing of a collection of independent events, e.g. applying an image-processing operation in parallel over a collection of images.
- **Classic MapReduce (PAV-2)**: Independent calculations (maps) followed by a final reduction step, e.g. WordCount from Big Data.
- **Map-Collective (PAV-3)**: Independent calculations (maps) with collective communications. For example, parallel machine learning is dominated by collectives such as scatter, gather, and reduce.
- **Map Point-to-Point (PAV-4)**: Independent computations with point-to-point communications. Graph analytics and simulations often employ local neighbor communications.
- **Map Streaming (PAV-5)**: Independent tasks with streaming data. This is seen in recent approaches to processing real-time data.
- **Shared Memory (PAV-6)**: In CDs we do not consider pure SM architectures but look at hybrids of SM and other DM models.

4.9.2. Facets in Execution View.

- **EV D4, D5, D6**: Represents 3V's out of the usual 4V's of Big Data [3].

- **EV M4**: An application whether Big Data or Big Simulation can be broken up into data plus model [38]. This facet identifies the size of the model, which is an important factor in application's performance.
- **EV M8**: Identifies the type and structure of communication such as collective, point-to-point, or pub/sub. This is another important aspect in performance.
- **EV M11**: Identifies the iterative nature of applications, which is common in parallel machine learning.
- **EV M14**: Complexity of the model.

4.9.3. Facets in Data Source and Style View. Facets in this are mostly self-explanatory but we like to highlight the followings.

- **Streaming (DSSV-D5)**: The streaming data has a lot of recent progress, especially within the Big Data community.
- **HPC Simulations (DSSV-D9)**: Identifies data that is generated as output of another program. In all of Scalable Parallel Interoperable Data Analytics Library (SPIDAL) analytics, the input to DA-MDS and Deterministic Annealing Pairwise Clustering (DA-PWC) comes in this form.

4.9.4. Facets in Processing View. Facets in the Processing View characterize algorithms for both Big Data and Big Simulations. These facets identify application's model only and do not represent data.

- **Micro-benchmarks (PV-M1)**: Represents functional benchmarks such as the kernel benchmarks in NPB.

- **Local (Analytics/Informatics/Simulations) (PV-M2)**: Identifies models dealing with only local data like those of neighbors.
- **Global (Analytics/Informatics/Simulations) (PV-M3)**: Represents models with global data synchronizations. For example, parallel machine learning typically requires global knowledge at each step.
- **PV 4M through 11M**: Facets for Big Data models.
- **PV 16M through 22M**: Facets for Big Simulation models.

Some facets in the CDs model may require extensions. For example, the 6M Data Search/Query/Index facet in the Processing View could be further divided into more facets to represent the type of query used. While this model is still in research and a proper classification of applications is necessary to evaluate its usefulness, its detailed faceted nature brings a systematic approach to classifying both Big Data and Big Simulations.

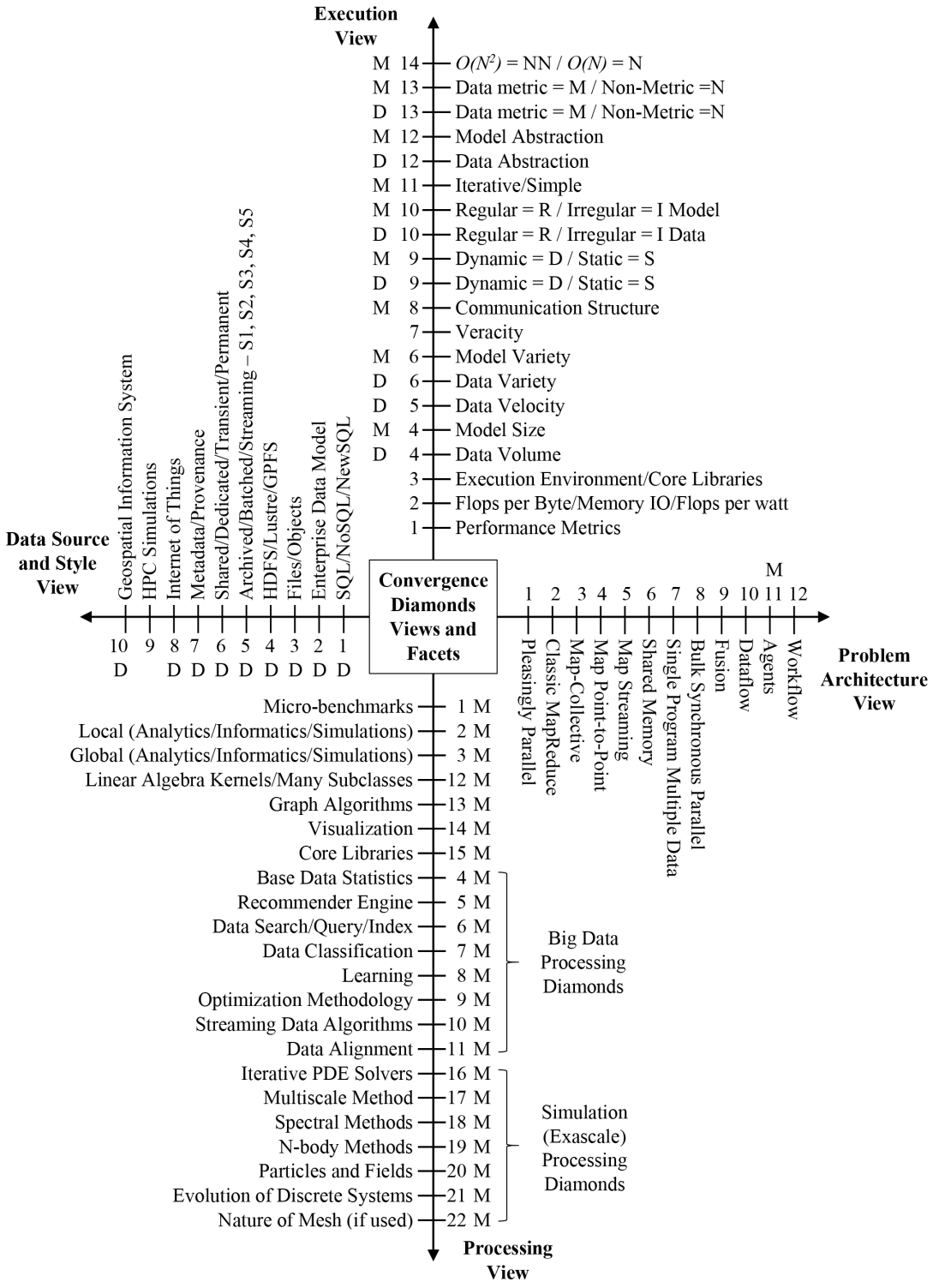


FIGURE 4.6. Dimensions and facets of CDs

CHAPTER 5

Performance Factors of Big Data

The emergence of Big Data poses numerous challenges from data storage to processing. While a plethora of benchmarks attempt to serve as yardsticks in this diverse landscape, we find it is imperative to identify the factors governing Big Data performance to perform a systematic study. It is also worth noting that most Big Data systems are written in Java, which brings out additional performance challenges beyond the nature of the Big Data application. The following sections identify three major issues: thread models, affinity patterns, and communication mechanisms as factors significantly affecting performance and show how to optimize them so that Java can match the performance of traditional HPC languages like C. Further, we look at four additional important aspects for achieving high performance with Java and Big Data. While these are discussed with respect to Java, they apply equally well to other Java-like, high-level languages such as Scala.

5.1. Thread Models

Threads offer a convenient construct for implementing shared memory parallelism. A common pattern used in both Big Data and HPC is the Fork-Join (FJ) thread model. In this approach a master thread spawns parallel regions dynamically as required. FJ regions are implicitly synchronized at the end, after which the worker threads are terminated and only the master thread will continue until a new parallel region is created. Thread creation and

termination are expensive, so FJ implementations employ thread pools to hand over forked tasks. Pooled threads are long-lived yet short-activated; i.e. they release CPU resources and switch to idle state after executing their tasks. This model is subsequently referred to as "Long Running Threads Fork-Join (LRT-FJ)" in this thesis. Java has built-in support for LRT-FJ through its `java.util.concurrent.ForkJoinPool` [1]. Habanero Java [**50**], an OpenMP-like [**21**] implementation in Java, also supports LRT-FJ via its `forall` and `forallChunked` constructs.

We experimented with another approach to shared memory parallelism, or Long Running Threads Bulk Synchronous Parallel (LRT-BSP). LRT-BSP resembles the classic Bulk Synchronous Parallel (BSP) style but with threads. Figure 5.1 depicts a side-by-side view of LRT-FJ and LRT-BSP models. The notable difference is that in LRT-BSP, threads are busy from start to finish of the program, not only within the parallel region as in LRT-FJ. The next important difference is the use of explicit synchronization constructs (blue horizontal lines in the figure) after non-trivial parallel work (red bars in the figure) in LRT-BSP. There are constructs such as `CyclicBarrier` in Java to aid the implementation of these synchronization steps. However, we employed native compare-and-swap (CAS) operations and busy loops for performance as well as to keep threads 'hot' on cores. A third difference in LRT-BSP is that the serial part of the code (green bars) is replicated across workers, whereas in LRT-FJ it is executed by just the master thread. Performance results show that the replication of serial work in LRT-BSP does not add significant overhead. The reason for this behavior is that in a well-designed parallel application, the serial portions are trivial

[1] https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

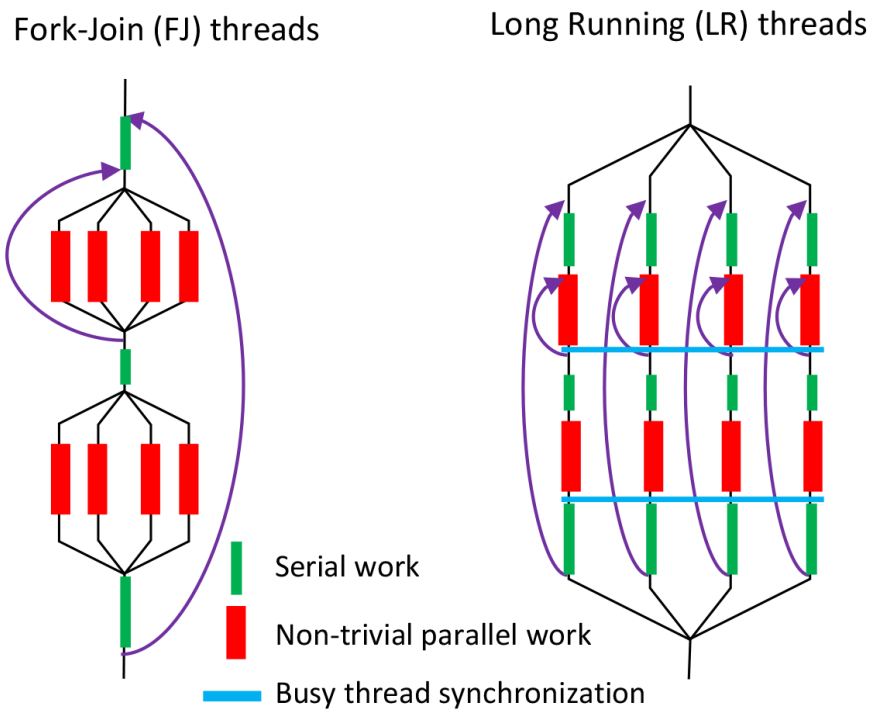


FIGURE 5.1. Fork-Join vs. long running threads

compared to the parallel work loads and the total amount of memory accesses in LRT-BSP is equal to that of LRT-FJ for these parts.

Beyond the differences in the execution model, we observed a significant performance improvement with LRT-BSP compared to LRT-BSP for parallel Java applications. Analyzing `perf` statistics revealed that LRT-FJ experiences a higher number of context switches, CPU migrations, and data translation lookaside buffer (dTLB) load/store misses than LRT-BSP. In an MDS run the factors were over 15x and 70x for context switches and CPU migrations, respectively. These inefficiencies coupled with the overhead of scheduling threads lead to noise in computation times within parallel FJ regions. Consequently, synchronization points

TABLE 5.1. Affinity patterns

		Process Affinity		
		Cores	Socket	None (All)
Thread Affinity	Inherit	CI	SI	NI
	Explicit per Core	CE	SE	NE

become expensive, and performance measurements indicate performance degradation as the number of threads increases in LRT-FJ.

5.2. Threads and Processes Affinity Patterns

Modern multicore HPC cluster nodes typically contain more than one physical CPU. Although memory is shared between these CPUs, memory access is not uniform. CPUs with their local memory compose NUMA domains or NUMA nodes. Developing parallel applications in these settings requires paying attention to the locality of memory access in order to improve performance.

In supported OSs process affinity determines where the OS can schedule a given process as well as the part of memory it can access. Threads spawned within a process inherit the affinity policy of the process by default. Also, it is possible to set affinity to threads explicitly, as desired for performance reasons. This research explores six affinity patterns and identifies binding options that produce the best and worst performance.

Details of the three process affinity patterns in Table 5.1 are:

Core: binds the process to N cores, where N is the number of threads used for shared memory parallelism.

Socket: binds the process to a physical CPU or socket.

None (All): binds the process to all available cores, which is equivalent to being unbound.

Worker threads may either inherit the process binding or be pinned to a separate core. K-Means and MDS performance tests revealed that selecting proper affinity settings out of these patterns can substantially improve overall performance.

5.3. Communication Mechanisms

Processes within a node offer an alternative approach to exploiting intra-node parallelism from that of threads. Long-running processes like those found in MPI programs avoid frequent scheduling overheads and other pitfalls common to short-activated threads. However, since nothing is shared across processes, a higher communication burden is incurred than with threads, especially when making collective calls. Increasing process count to utilize all cores on modern chips with higher core counts makes this effect even worse, degrading any computational advantages of using processes.

Figure 5.2, for example, plots arithmetic average, hereafter referred to as "average", of MPI `allgatherv` times over 50 iterations against varying number of processes within a node. Note that all MPI implementations used default settings aside from the use of Infiniband transport. The MPI `allgatherv` test was a micro-benchmark based on the popular OSU Micro-benchmarks Suite (OMB) [**74**].

The purple and black lines show C implementations compiled against OpenMPI and MVAPICH2 [**49**], while the green line shows the same program in Java compiled against OpenMPI's Java binding. The Java binding is a thin wrapper around OpenMPI's C implementation. All tests used a constant 24 million bytes or three million double values. The HPC cluster used to run these experiments, Juliet, has 24 cores per node, so there are

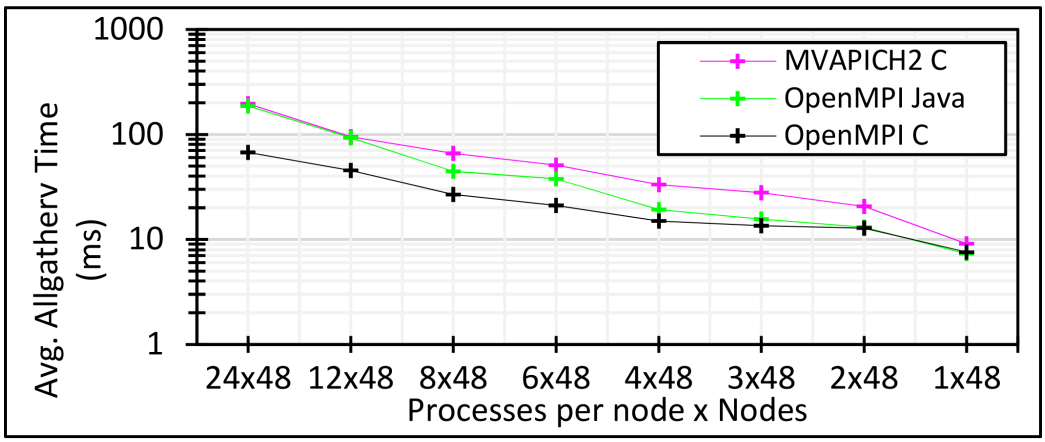


FIGURE 5.2. MPI `allgatherv` performance with different MPI implementations and varying intra-node parallelisms

eight combinations of threads and processes to yield the full 24-way parallelism within a node. The process numbers on the x-axis of Figure 5.2 correspond to those of the eight patterns. For example, eight processes imply that each has three threads internally for computations in a real parallel program such as DA-MDS or DA-PWC. In this experiment, however, threads were not used, as there was no computation necessary to benchmark communication performance.

The experiment shows that the communication cost becomes significant as the number of processes per node increases and that the effect is independent of the choice of MPI implementation and the use of Java binding in OpenMPI. However, an encouraging discovery is that all implementations produce a nearly identical performance for the single-process-per-node case. This led to the development of shared-memory-based, inter-process communication in Java as discussed below to reduce communication overhead.

Java shared-memory communication uses a custom memory maps implementation from OpenHFT's JavaLang [**72**] project to perform inter-process communication for processes

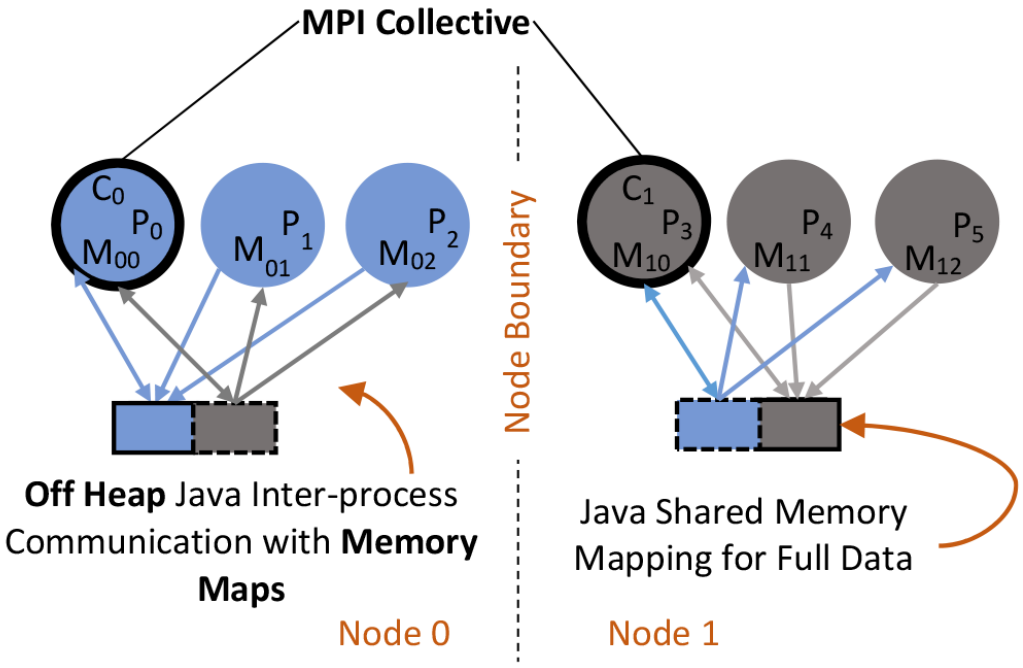


FIGURE 5.3. Intra-node message passing with Java shared memory maps

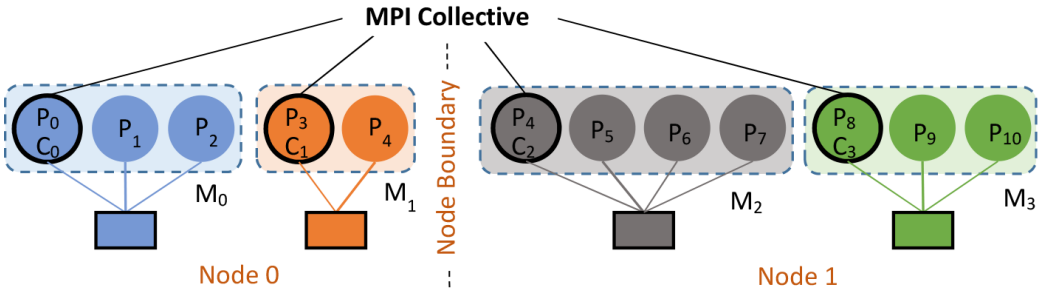


FIGURE 5.4. Heterogeneous shared memory intra-node messaging

within a node, thus eliminating any intra-node MPI calls. The standard MPI programming would require $O(R^2)$ of communications in a collective call, where R is the number of processes. In this optimization we have effectively reduced this to $O(\hat{N}^2)$, where $\hat{N}$ is the number of nodes, significantly less than R.

Figure 5.3 shows the general architecture of this optimization, where two nodes, each with three processes, are shown as an example. Process ranks range from P_0 to P_5 and

belong to `MPI_COMM_WORLD`. One process from each node acts as the communication leader - C_0 and C_1. These leaders have a separate MPI communicator called `COLLECTIVE_COMM`. Similarly, the processes within a node belong to a separate `MMAP_COMM`, for example M_{00} to M_{02} in one communicator for Node 0 and M_{10} to M_{12} in another for Node 1. Also, all processes within a node map the same memory region as an off-heap buffer in Java and compute necessary offsets at the beginning of the program. The takeaway point of this setup is the use of memory maps to communicate between processes and the reduction in communication calls. With this setup, a call to an MPI collective will follow the steps below within the program.

(1) All processes, P_0 to P_5, write their partial data to the mapped memory region, offset by their rank and node. See the downward blue arrows for Node 0 and gray arrows for Node 1 in the figure.

(2) Communication leaders, C_0 and C_1, wait for the peers, $\{M_{01}, M_{02}\}$ and $\{M_{10}, M_{11}\}$ to finish writing. Note that leaders wait only for their peers in the same node.

(3) Once the partial data is written, the leaders participate in the MPI collective call with partial data from their peers, the upward blue arrows for Node 0 and gray arrows for Node 1. Also, the leaders may perform the collective operation locally on the partial data and use its results before the MPI communication depending on the type of collective required. MPI `allgatherv`, for example, will not have any local operation to be performed, but a collective operation like `allreduce` benefits from doing the reduction locally. Note, the peers wait while their leader performs MPI communication.

(4) At the end of the MPI communication, the leaders write the results to the respective memory maps, downward gray arrows for Node 0 and blue arrows for Node 1. This data is then immediately available to their peers without requiring further communication, upward gray arrows for Node 0 and blue arrows for Node 1.

This approach reduces MPI communication to just two processes, in contrast to a typical MPI call, where six processes would be communicating with each other. The two wait operations mentioned above are implemented using Java atomic variables. They could also be implemented using an MPI `barrier` on the `MMAP_COMM`. This would cause intra-node messaging within the `barrier` call, which is negligible compared to the actual data communication. However, experiments showed that the approach using Java atomic variables is more efficient than using barriers.

The shared-memory implementation also supports heterogeneous rank distribution and multiple shared memory groups, as illustrated in Figure 5.4. These modes are described below.

Non-uniform rank distribution. In some HPC clusters it is possible to have node groups with different core counts per node. For example, Juliet is a modern Intel Haswell-based HPC cluster that we used for testing, which has two groups of nodes with 24 and 36 cores per node. In such situations it is possible to spawn a non-uniform number of processes per node using MPI and this particular shared memory communication implementation will automatically detect such heterogeneous configuration and adjusts its shared memory buffers accordingly.

Multiple memory groups per node - If more than one memory map per node (M) is necessary for some reason such as performance testing, this implementation will support additional communication groups to be created using a parameter. Figure5.4 shows two memory maps per node. As a result, $O(\hat{N}^2)$ communication is now changed to $O((\hat{N}M)^2)$, so it is highly recommended to use a smaller M, ideally $M = 1$.

5.4. Other Factors

While the above factors are the most important, they alone may not be enough to reach desirable performance with Big Data and Java. The following sections identify some of the other factors that are hard to completely avoid, yet can be minimized with careful design and the use of right libraries within the program.

5.4.1. Garbage Collection (GC). GC is an integral part of high-level languages like Java and is a convenient feature for programmers so they can develop applications without being concerned about memory management. However, despite the convenience, frequent and long GCs are expensive for performance-sensitive applications like those in the SPIDAL suite.

In Java GC works by segmenting the program's heap into regions called "generations" and moving objects between these regions depending on their longevity. Every object starts in the Young Generation (YG) and gets promoted to the Old Generation (OG) if they have lived long enough. Minor garbage collections happen in the YG frequently and short-lived objects are removed without GC going through the entire heap. Also, the surviving long-lived objects in the YG are moved to the OG. When the OG has reached

its maximum capacity, a full GC happens, which is an expensive operation depending on the size of the heap and can take a considerable amount of time. Moreover, both minor and major collections stop all the running threads within the process to make sure data consistency while moving objects. Such GC pauses incur significant delays, especially for machine-learning-like applications where slowness in one process affects all others since they must synchronize global communications.

While in most cases it is impractical to avoid GC completely, it is vital to keep GC at a minimum to improve performance. At the same time the memory footprint should be minimized in order to scale over larger data sizes. The following techniques can be used both to lower the GC frequency and memory footprint.

Object Reuse. One of the best ways to reduce GC frequency is to reduce the creation of new temporary objects. A simple example of the frequent creation of temporary objects is string concatenation. Java string objects are immutable, meaning any concatenation will create a whole new string object that contains both the previous and new values. If there is no reference to the previous string object, then it will become garbage. Similarly, temporary arrays created within loops are another common garbage generation pattern. Object reuse allows the reduction of such garbage production by reusing the same allocated memory in the heap to store updated values. For example, `StringBuffer` in Java provides a reusable array-based string buffer, which produces garbage only when array capacity is exceeded. Moreover, object reuse is helpful not only when dealing with objects that are expensive to create but also with smaller, less expensive objects. In the latter case object reuse saves the heap from becoming fragmented due to frequent smaller object allocations.

Static Allocation. Static allocation complements object reuse, as allocations are made on a per-class basis and not a per-object basis. This technique is useful when allocating data structures for common operations like holding the result of matrix multiplication or to load initial input data. SPIDAL algorithms use this technique extensively to make one-time memory allocations for all their common data structures.

5.4.2. Object Serialization and Deserialization. It is unavoidable, as well as prohibitively expensive, to serialize and deserialize objects in Java. The default Java serialization is too verbose to use for any performance-oriented work. One option to solve this is to use an efficient serialization library like Kryo [**88**]. Kryo produces a compact binary representation of objects. A comparison of serialization libraries can be found in the JVM Serializers page [**86**]. The other option to avoid serialization costs is to model data structures around Java direct buffers [**18**]. These byte buffers are allocated outside the GC managed heap. Objects represented by using such buffers do not need explicit serialization. The caveat is that the programmer has to implement custom object representations, and it is cumbersome to make changes.

5.4.3. Memory References and Cache. The OOP style provides a rich environment to create arbitrarily nested objects. A pitfall to OOP is the increase in number of indirect memory references that the underlying JVM has to make when accessing data elements. For example, consider accessing a two-dimensional array in a classic nested loop where the outer loop run across rows and inner one goes through columns. If we were to name the array `A` and the loop indices **i** and **j** for the outer and inner loops, then from our experience accesses of the form `A[i][j]` are more expensive than if `A` were a single-dimensional array

and accessed as `A[i*numColumns+j]`, where `numColumns` is the number of columns. Also, if two-dimensional arrays are unavoidable, then caching the row corresponding to the outer loop as `rowA = A[i]` and using `rowA` inside the inner loop is more efficient than directly accessing `A` as `A[i][j]`.

5.4.4. Data Read Write. The default approach to read and write data in Java has been to use the input and output streams **[17]**. As the name implies, these read data as a stream, which is a bottleneck when reading large amounts of data. An efficient alternative is to use bulk loading with Java memory-mapped files. Memory mapping directly maps a given number of bytes into memory and returns a direct buffer to access the mapped content. These buffers exist outside the GC-managed heap, which makes them ideal for other input and output operations without requiring additional copies. SPIDAL applications use memory-mapped files for both initial data loading and to communicate with other processes.

CHAPTER 6

Performance Evaluation

This chapter looks at performance improvements for the three major factors introduced in the previous chapter using two of SPIDAL's algorithms: DA-MDS and K-Means clustering. We also present performance of DA-PWC and MDSasChisq to highlight the fact that they too exhibit poor performance in the absence of the performance improvements discussed in Chapter 5. All tests were performed on a modern HPC cluster, Juliet, which is a 128-node Intel Haswell cluster. Of the 128 nodes, 96 nodes have 24 cores (2 sockets x 12 cores each), and 32 nodes have 36 cores (2 sockets x 18 cores each) per node, totaling 3,456 cores. Each node consists of 128GB of main memory and 56 gigabits per second (Gbps) Infiniband interconnect. We explicitly look at scaling within a node and across nodes of these applications.

6.1. Performance of K-Means Clustering

We have implemented six variants of K-Means clustering in this performance study. Four of them are OpenMPI-based in both Java and C, supporting LRT-FJ and LRT-BSP thread models. The remainder are based on Flink and Spark. Details of these implementations are as follows.

6.1.1. MPI Java and C K-Means. The two C implementations use OpenMPI for message passing and OpenMP for shared memory parallelism. LRT-FJ follows the conventional MPI plus `#pragma omp parallel` regions. LRT-BSP, on the other hand, starts an OpenMP parallel region after `MPI_INIT` and continues to follow the models illustrated in Figure 5.1. Intermediate thread synchronization is done through atomic built-ins of the GNU Compiler Collection (GCC). The source code for LRT-FJ and LRT-BSP is available in GitHub [**28**] under branches `master` and `lrt`, respectively.

The Java implementations use OpenMPI's Java binding [**73**, **92**] and Habanero-Java [**50**] thread library, which provides similar parallel constructs to OpenMP. In LRT-BSP intermediate thread synchronization uses Java atomic support, which is more efficient than other lock mechanisms in Java. The source code for LRT-FJ and LRT-BSP is available in GitHub [**29**] under branches `master` and `lrt-debug`, respectively.

6.1.2. Flink K-Means. Flink is a distributed dataflow engine for Big Data applications and provides a dataflow-based programming and execution model. The dataflow computations composed by the user are converted to an execution dataflow graph by Flink and executed on a distributed set of nodes.

Flink's K-Means [**54**] dataflow graph is shown in Figure 6.1. Inputs to the algorithm are a set of points and a set of centroids read from the disk. At each iteration a new set of centroids is calculated and fed back to the beginning of the next iteration. The algorithm partitions the points into multiple map tasks and uses the full set of centroids in each map task. Each map task assigns its points to their nearest centroid. For each centroid, the average of such points is reduced (summed) to get the new set of centroids, which is

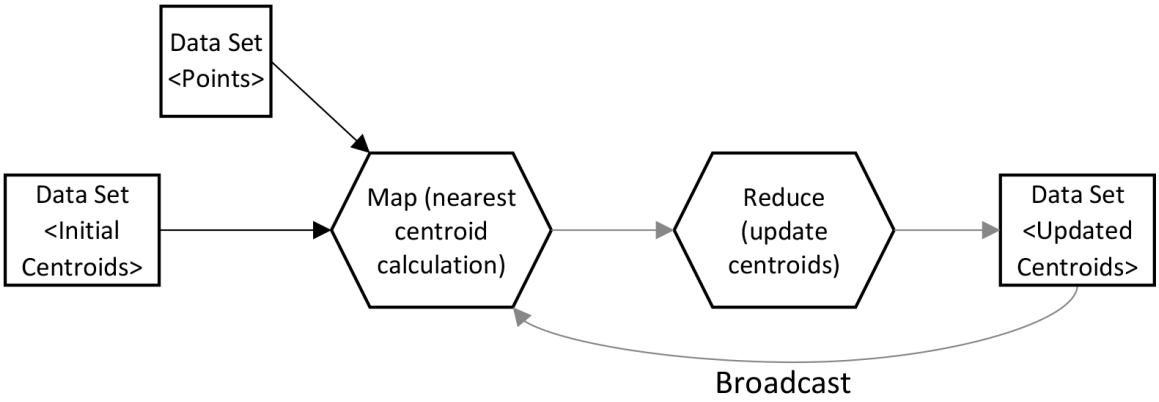


FIGURE 6.1. Flink and Spark K-Means algorithm. Both Flink and Spark implementations follow the same data-flow

broadcast to the next iteration. This is essentially the same algorithm as that used in MPI but expressed as a stream of dataflow transformations. In particular the Flink reduction and broadcast are equivalent to `MPI_Allreduce` semantics.

6.1.3. Spark K-Means. Spark is a distributed in-memory data processing engine. The data model in Spark is based around RDDs [**97**]. The execution model of Spark is based on RDDs and lineage graphs. The lineage graph captures dependencies between RDDs and their transformations. The logical execution model is expressed through a chain of transformations on RDDs by the user.

We used a slightly modified version [1] of the K-Means implementation provided in the Spark MLlib [**69**] library . The overall dataflow is shown in Figure 6.1, which is the same as that of Flink K-Means, and the inputs are read in a similar fashion from disk. The points data file is partitioned and parallel map operations are performed on each partition. Each point in a data partition is cached to increase performance. Within the map operations,

[1] https://github.com/DSC-SPIDAL/spark/tree/1.6.1.modifiedKmeans

points are assigned to their closest centers. The reduce step gathers all this information to the driver program where the new set of centers are calculated and broadcast to all the worker nodes for the next iteration.

6.1.4. Evaluation. Figure 6.2 and Figure 6.3 show K-Means Java and C total runtime for 1 million 2D points and 1,000 centroids, respectively. Each figure presents performance of both LRT-FJ and LRT-BSP models over the six binding patterns identified in Table 5.1. These were run on 24-core nodes; hence the abscissa shows all eight possible combinations of threads and processes within a node that produce the 24-way parallelism, which utilizes all cores of the particular node. The left-most pattern, 1x24, indicates all processes, and the right-most pattern, 24x1, indicates all threads within a node. Note that patterns 8x3 and 24x1 imply that processes span across NUMA memory boundaries, which is known to be inefficient but is presented here for completeness. The red and orange lines represent inherited thread affinity for LRT-FJ and LRT-BSP, respectively. Similarly, the black and green lines illustrate explicit thread pinning, each thread to a core, for these two thread models.

The Java results suggest LRT-FJ is the worst in performance whatever the affinity strategy for any pattern other than 1x24, which is all MPI and does not use thread-parallel regions. A primary reason for this poor performance is the thread scheduling overhead in Java, as FJ threads are short activated. Also, the JVM spawns extra bookkeeping threads for GC and other tasks, which compete for CPU resources as well. Of the LRT-BSP lines the unbound threads (`NI`) show the worst performance. Affinity patterns `NE` and `CE` seem to give the best runtime as the number of threads increases.

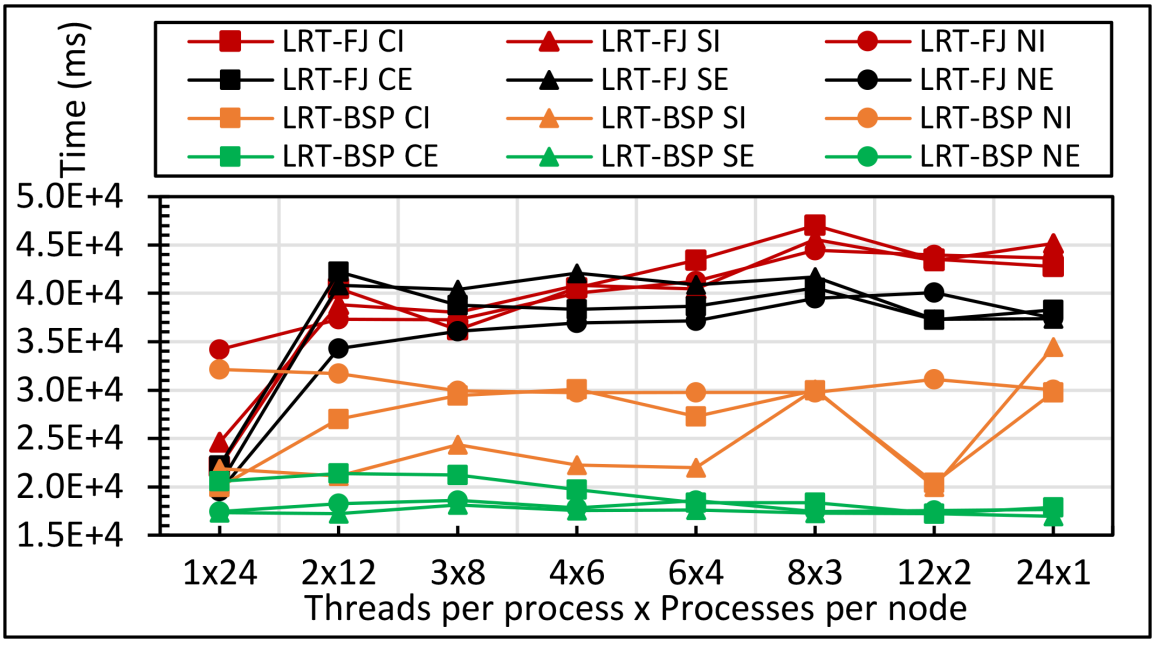


FIGURE 6.2. Java K-Means 1 mil points and 1k centers performance on 16 nodes for LRT-FJ and LRT-BSP with varying affinity patterns over varying threads and processes.

The C results show the same behavior for unbounded and explicitly bound threads. The two thread models, however, show similar performance, unlike Java. Further investigation of this behavior revealed that OpenMP threads keep the CPUs utilization at 100% between FJ regions, which suggests that OpenMP internally optimizes threads similar to the Java LRT-BSP implementation introduced in this paper.

Figure 6.4 illustrates the effect of affinity patterns `CE` and `NE` for varying data sizes on LRT-BSP. These performed similarly, but the results suggested `CE` is better than `NE`.

Figure 6.5 compares Java and C LRT-BSP runtimes for K-Means over varying data sizes across thread and process combinations. Results demonstrate that Java performance is on par with C. Also, sometimes Java outperforms C, mostly due to Just In Time (JIT) optimizations, as seen in the figure for 500k centers.

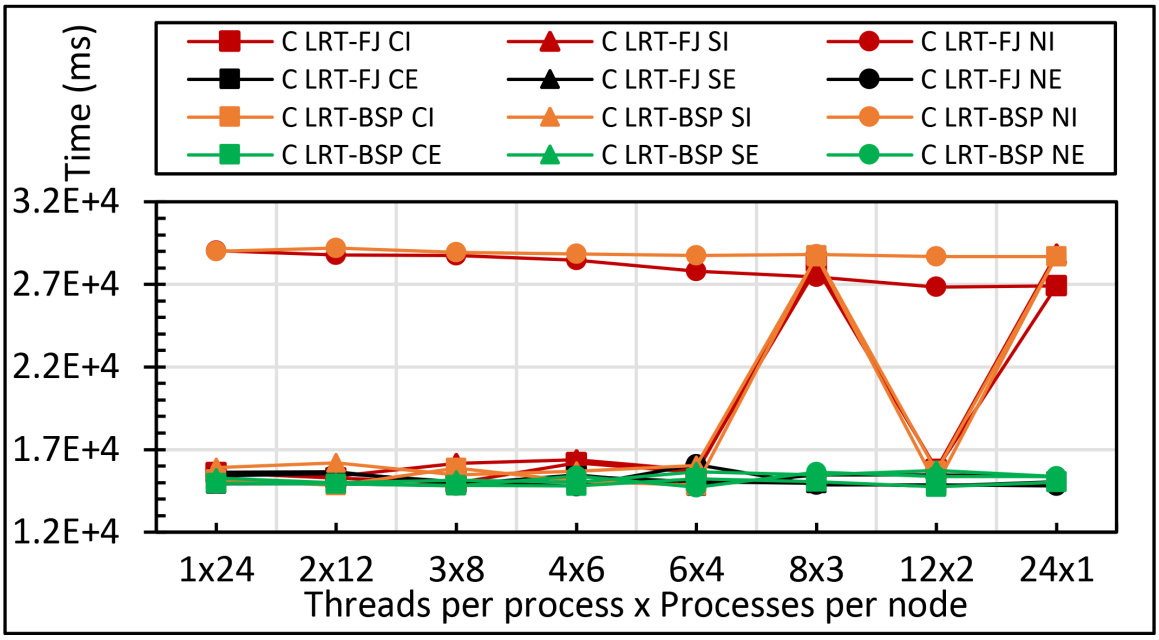


FIGURE 6.3. C K-Means 1 mil points and 1k centers performance on 16 nodes for LRT-FJ and LRT-BSP with varying affinity patterns over varying threads and processes.

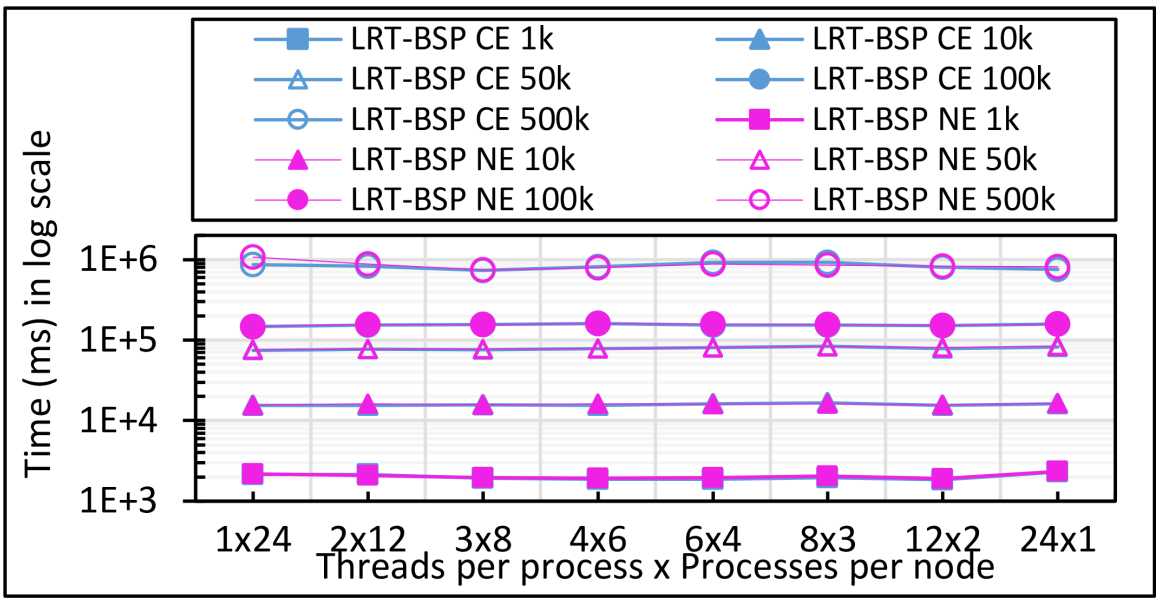


FIGURE 6.4. Java K-Means LRT-BSP affinity CE vs NE performance for 1 mil points with 1k,10k,50k,100k, and 500k centers on 16 nodes over varying threads and processes.

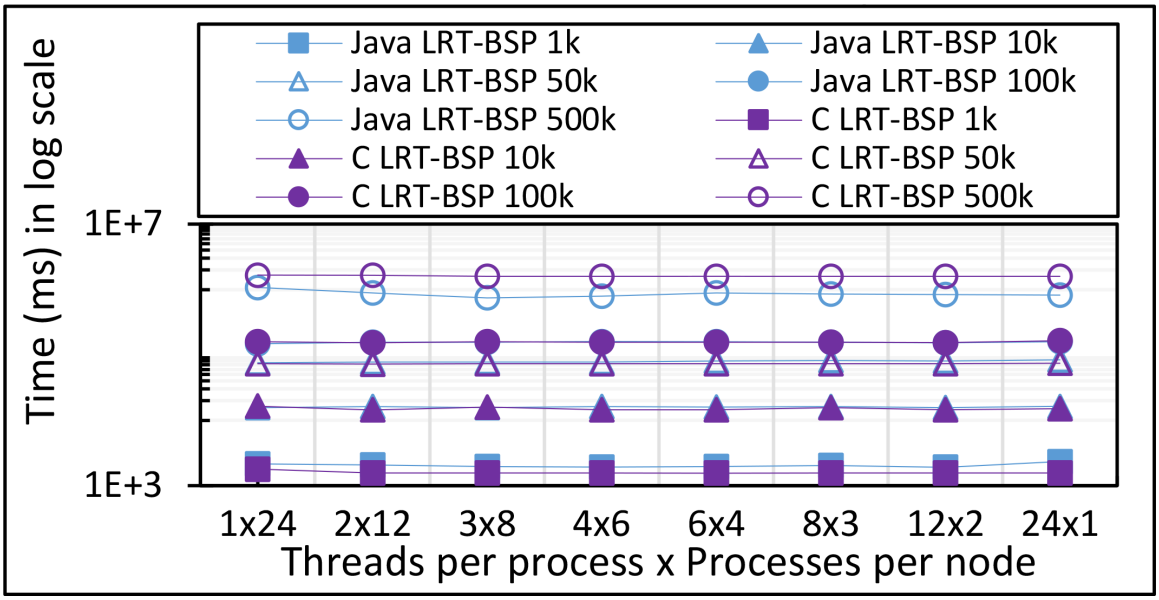


FIGURE 6.5. Java vs C K-Means LRT-BSP affinity CE performance for 1 mil points with 1k,10k,50k,100k, and 500k centers on 16 nodes over varying threads and processes.

Figure 6.6 and Figure 6.7 showcase LRT-FJ and LRT-BSP performance over varying data sizes for affinity pattern CE. In Figure 6.6 the experiment was carried out for increasing number of centroids as 1k,10k, and 100k. LRT-BSP shows constant performance across thread and process combinations for all data sizes, whereas LRT-FJ exhibits abysmal performance as thread and data sizes increase. Figure 6.7 replicates the same experiment for data sizes 50k and 500k. Again, the results agree with those of Figure 6.6.

Hitherto the K-Means performance charts have investigated the behavior of different combinations of threads and processes with LRT-FJ and LRT-BSP thread models in Java and C over varying data sizes and affinity patterns. The total parallelism within a node was kept constant at 24, which is the maximum parallelism possible for the particular test nodes in Juliet. Figure 6.8, in contrast, explores the speedup of these different models with increasing intra-node parallelism. The green line is Java with all processes, each pinned to a separate

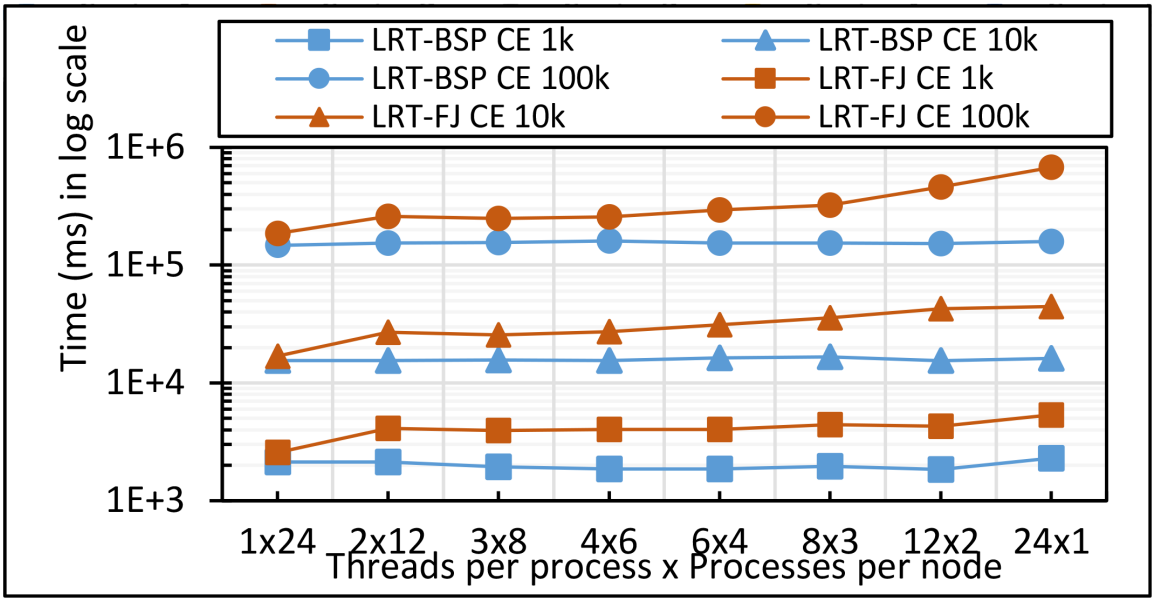


FIGURE 6.6. Java K-Means 1 mil points with 1k,10k, and 100k centers performance on 16 nodes for LRT-FJ and LRT-BSP over varying threads and processes. The affinity pattern is CE.

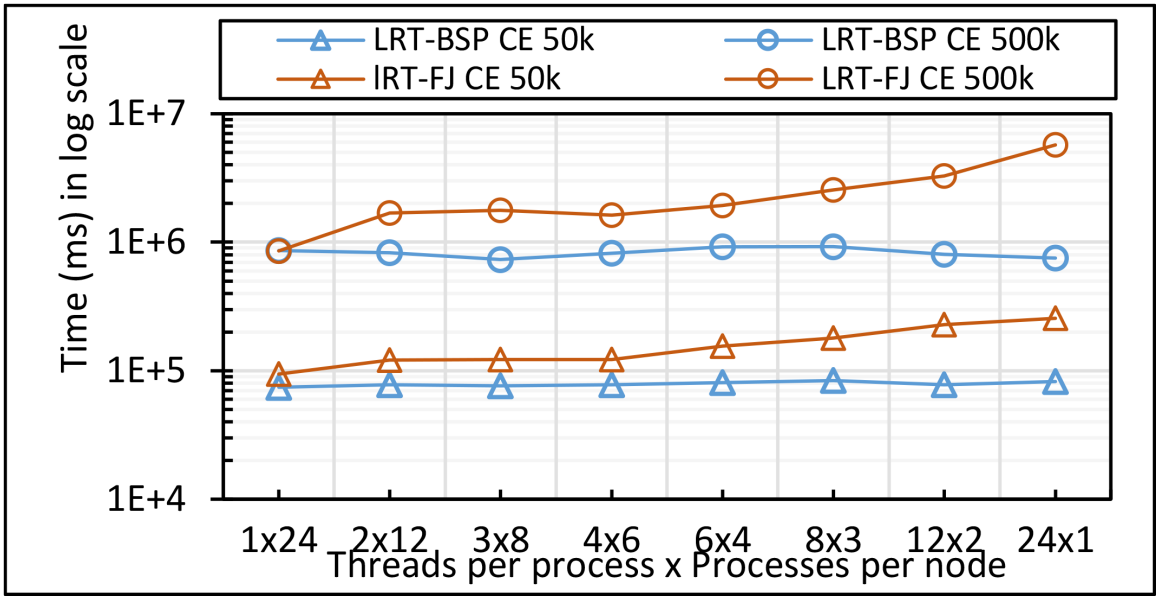


FIGURE 6.7. Java K-Means 1 mil points with 50k, and 500k centers performance on 16 nodes for LRT-FJ and LRT-BSP over varying threads and processes. The affinity pattern is CE.

core in each machine. The purple line is Java LRT-BSP with intra-node parallelism and is all threads. The implication of this is that for parallelisms 16 and 24, threads would span across NUMA boundaries as these Juliet nodes only have 12 cores per socket. This is known to be inefficient, hence the dashed purple line with black triangles showing the hybrid use of processes and threads for these two cases. For example, the 24-way parallelism was run as 12x2, where two processes were run pinned to each socket and 12 threads were spawned within each process pinned to each core of a particular socket. Similarly, 16-way parallelism was done as 8x2. The orange lines show Java LRT-FJ, again the black triangles showing the hybrid approach. All speedup values are based on the single process per node case of the green line.

The results of this experiment suggest that the performance of Java LRT-BSP threads is on par with that of all processes, whereas LRT-FJ shows abysmal performance. Figure 6.8 also includes the C LRT-FJ performance using OpenMP threads. Unlike Java FJ the OpenMP's FJ regions appear to implement an optimized version similar to Java's LRT-BSP. This agrees with the earlier comparison of Java and C in Figure 6.3. It also is worth noting how Java performance is competitive with C and produces near linear speedup.

The remainder of K-Means evaluation focuses on comparing Spark and Flink against MPI. The evaluation was done in 16 nodes, each with 24 cores. We measured the difference between total time and computation time to estimate overheads including communication. Note that in both Spark and Flink communications are handled internally to the framework, and it is not possible to measure the cost of communications through the available API functions. The results are shown in Figure 6.9 for 1 million 2D data points with varying

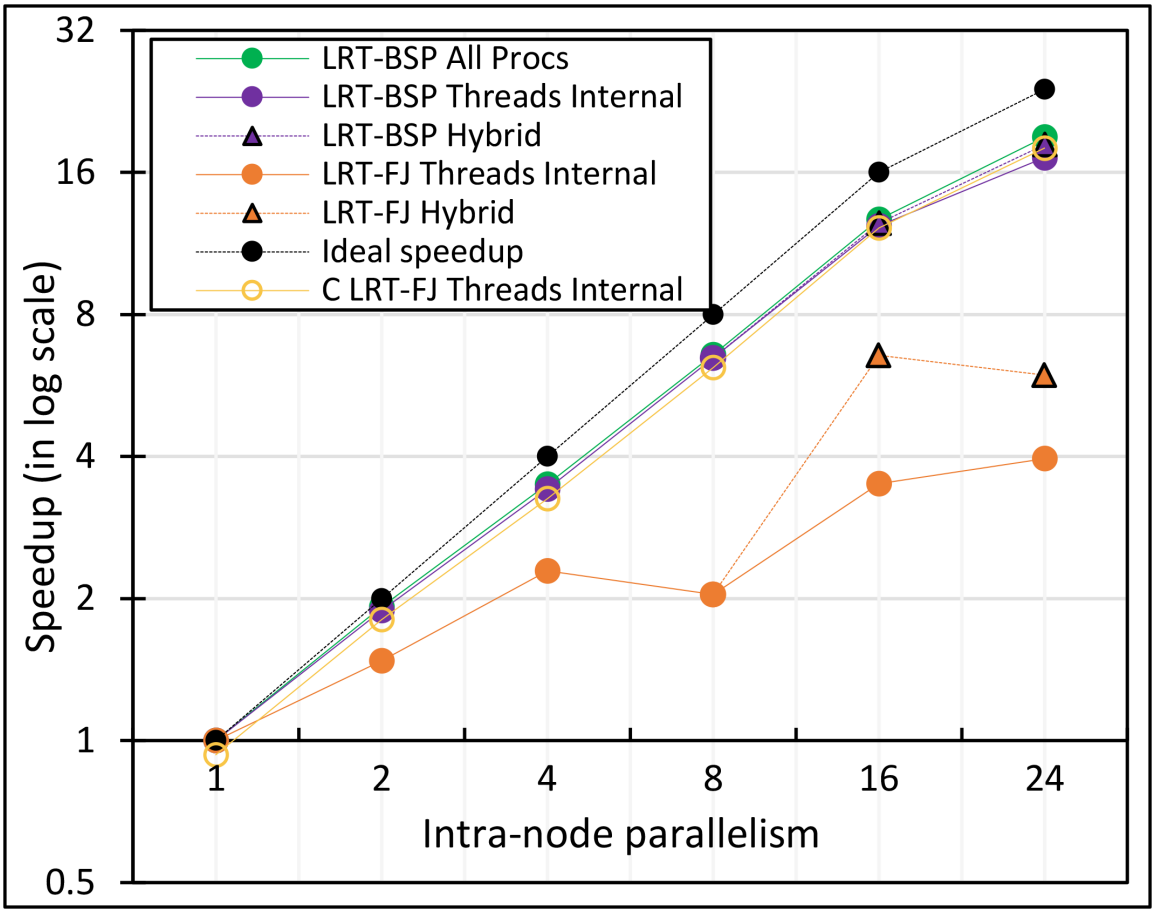


FIGURE 6.8. Java and C K-Means 1 mil points with 100k centers performance on 16 nodes for LRT-FJ and LRT-BSP over varying intra-node parallelisms. The affinity pattern is CE.

number of centroids. We observed significant communication overhead in these frameworks compared to MPI. The primary reason for such poor performance is the sub-optimal implementation of reductions in Flink and Spark.

Figure 6.10 illustrates the dataflow reduction model implemented in Spark and Flink, where all parallel tasks send data to a single or multiple reduce tasks to perform the reduction. K-Means requires an MPI like `Allreduce` semantics; hence the reduction in these programs is followed by a broadcast. Similar to the reduction operation, the broadcast

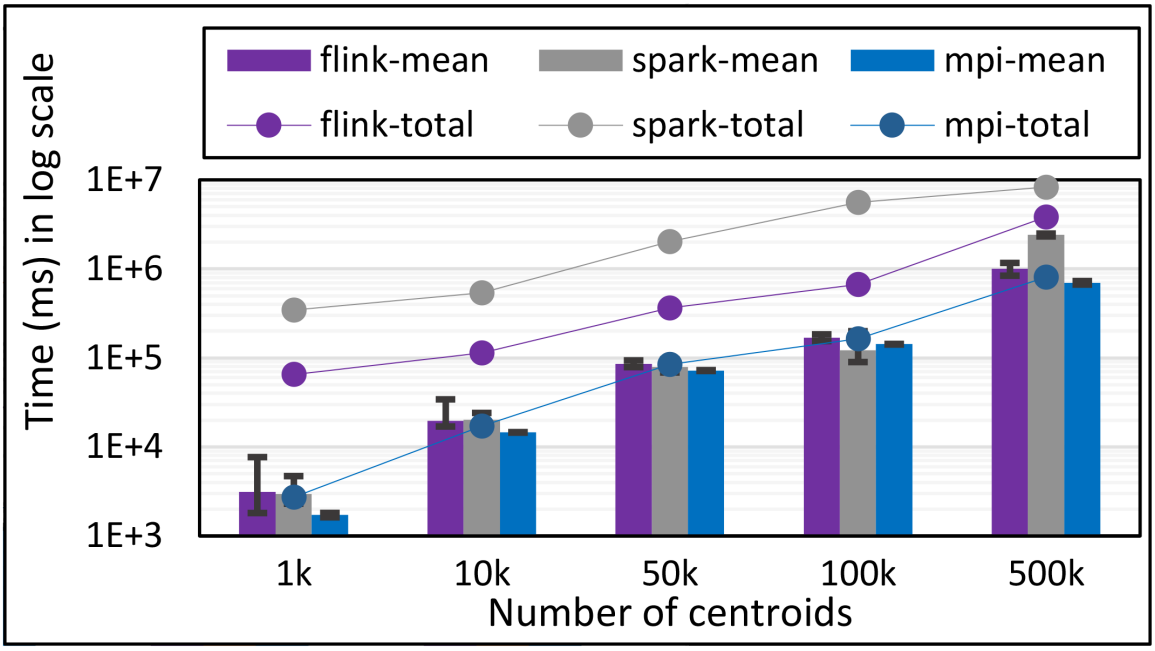


FIGURE 6.9. K-Means total and compute times for 1 million 2D points and 1k,10,50k,100k, and 500k centroids for Spark, Flink, and MPI Java LRT-BSP CE. Run on 16 nodes as 24x1.

is implemented serially as well. As the number of parallel tasks and the message size increase, this two-step approach becomes highly inefficient in performing global reductions. On the other hand, MPI uses a recursive doubling algorithm for doing the reduction and broadcast together, which is very efficient and happens in-place.

Since the communication overhead was dominant in K-Means algorithm, we performed a single node experiment with one process and multiple threads to look at computation costs more closely. With one process there is no network communication in Flink or Spark and Figure 6.11 illustrates the results. Flink uses an actor-based execution model, which uses the Akka [**45**] framework to execute tasks. The framework creates and destroys LRT-FJ-style threads to execute the individual tasks. Spark uses an executor/task model where an executor creates at most a single task for each of its allocated cores. With this experiment we

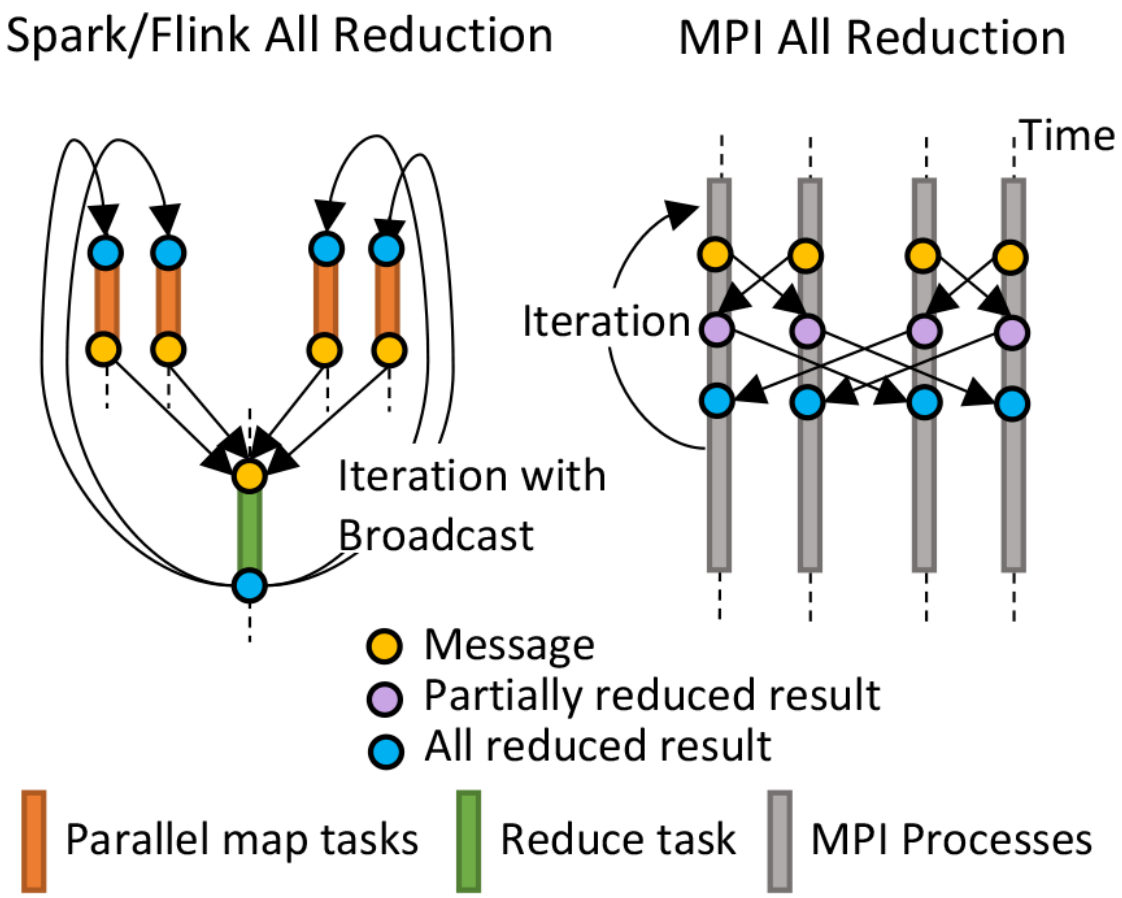


FIGURE 6.10. Spark and Flink's all reduction vs MPI all reduction.

have observed execution time imbalances among the parallel tasks for both Spark and Flink. The same has been observed with the LRT-FJ Java MPI implementation of K-Means, and we could minimize these effects in MPI Java with the LRT-BSP style executions. Balanced parallel computations are vital to efficient parallel algorithms as the slowest task dominates the parallel computation time.

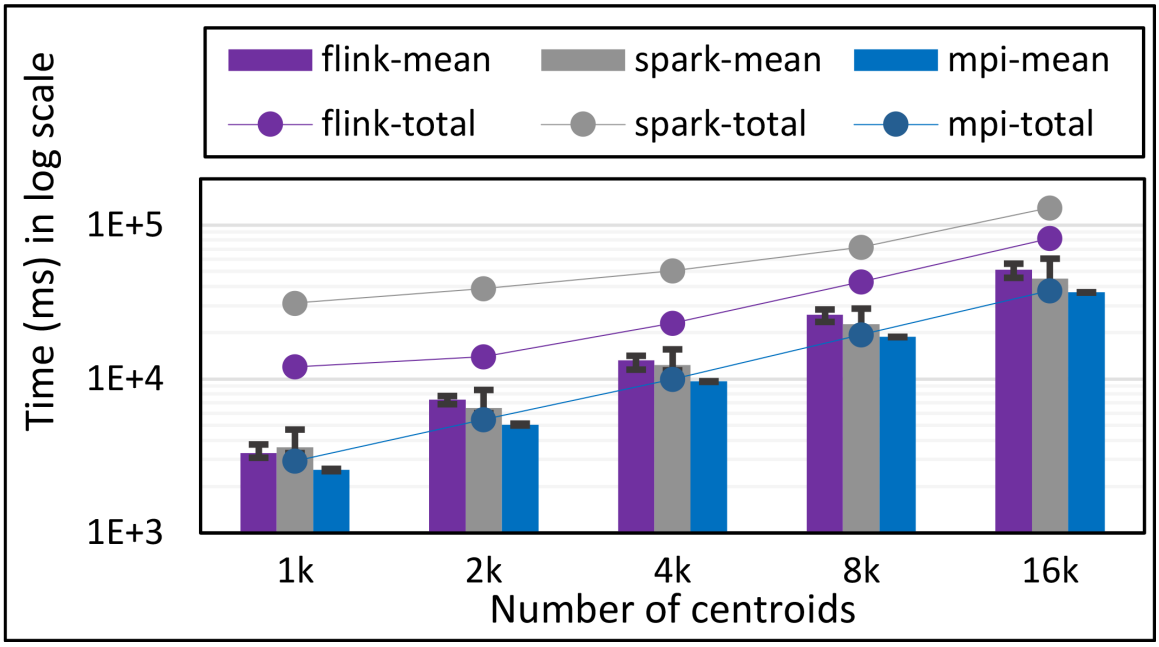


FIGURE 6.11. K-Means total and compute times for 100k 2D points and 1k,2k,4k,8k, and 16k centroids for Spark, Flink, and MPI Java LRT-BSP CE. Run on 1 node as 24x1

6.2. Performance of DA-MDS

To evaluate DA-MDS we have implemented two versions for LRT-FJ and LRT-BSP models. Both these flavors include shared memory communication and other optimizations discussed in Section 5.4 and in the SPIDAL Java **[30]** paper. Computations in DA-MDS grow $O(N^2)$ and communications $O(N)$. Also, unlike K-Means, where only one parallel region is required, DA-MDS requires multiple parallel regions revisited on each iteration until converged. This hierarchical iteration pattern (parallel conjugate gradient iteration inside a classic expectation maximization loop) causes issues with the Big Data frameworks that we will discuss in a latter section.

6.2.1. Evaluation. Figure 6.12 through Figure 6.17 illustrate DA-MDS performance for data sizes 50k, 100k, and 200k on 24-core and 36-core nodes. Each figure presents DA-MDS runtime for the two thread models and affinity patterns `CE`, `SE`, `NE`, and `NI`. Patterns `CI` and `SI` were omitted as they showed similar abysmal performance as `NI` in earlier K-Means results. Thread and process combinations for 24-core nodes were the same as those used in K-Means experiments. On 36-core nodes, nine patterns were tested from 1x36 to 36x1. However, as LRT-BSP allocates data for all threads at the process level, 200k decomposition over 16 nodes produced more data than the Java 1D arrays could hold. Therefore, this pattern could not be tested for 200k data. LRT-BSP did not face this situation, as data structures are local to threads and each allocates only the data required for the thread, which is within Java's array limit of $2^{31} - 1$ elements.

The above results confirm that Java LRT-FJ has the lowest performance irrespective of the binding, data size, or the number of threads. On the other hand the LRT-BSP model produced constant high performance across all these parameters. Investigating these effects further, an 18x2 run for 100k data produced the `perf` stats in Table 6.1, which show a vast number of context switches, CPU migrations, and data translation lookaside buffer load misses for LRT-FJ compared to LRT-FJ. These statistics are directly related with performance and hence explain the poor performance of LRT-FJ model.

Table 6.2 presents scaling of DA-MDS across nodes for data sizes 50k, 100k, and 200k. Speedup values are measured against the all-process – 1x24 or 1x36 – base case. Performance is expected to double as the number of nodes doubles. However, none of the 12x2 LRT-FJ values came close to the expected number. These are shown in red. In contrast 12x2 of

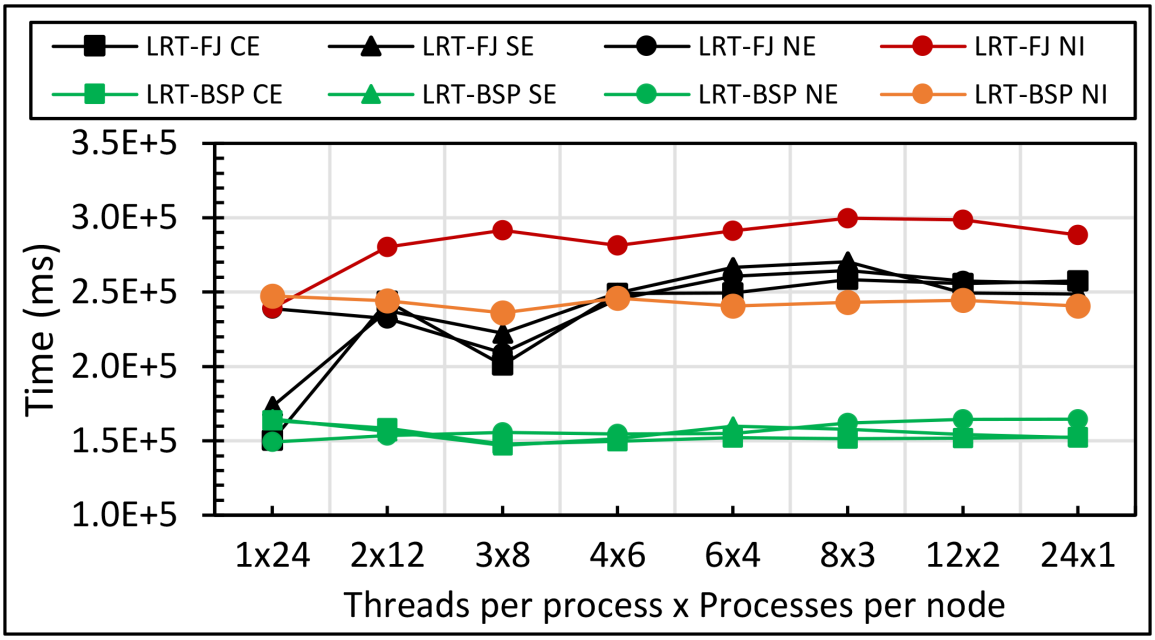


FIGURE 6.12. Java DA-MDS 50k points performance on 16 nodes for LRT-FJ and LRT-BSP over varying threads and processes. Affinity patterns are CE,NE,SE, and NI.

TABLE 6.1. Linux `perf` statistics for DA-MDS run of 18x2 on 32 nodes. Affinity pattern is `CE`.

	LRT-FJ	**LRT-BSP**
Context Switches	477913	31433
CPU Migrations	63953	864
dTLB load misses	17226323	6493703

LRT-BSP follows the expected doubling in performance and also can produce slightly better results than 1x24 as data increases.

The previous K-Means results and the DA-MDS explored the effects of thread models and affinity patterns in detail. The following DA-MDS tests focus on communication mechanisms and other factors mentioned in Section 5. Figure 6.18 through Figure 6.20 illustrate DA-MDS performance for 100k, 200k, and 400k data sizes over varying threads and processes combinations. With $O(N^2)$ growth in runtime, a 400k test would take four

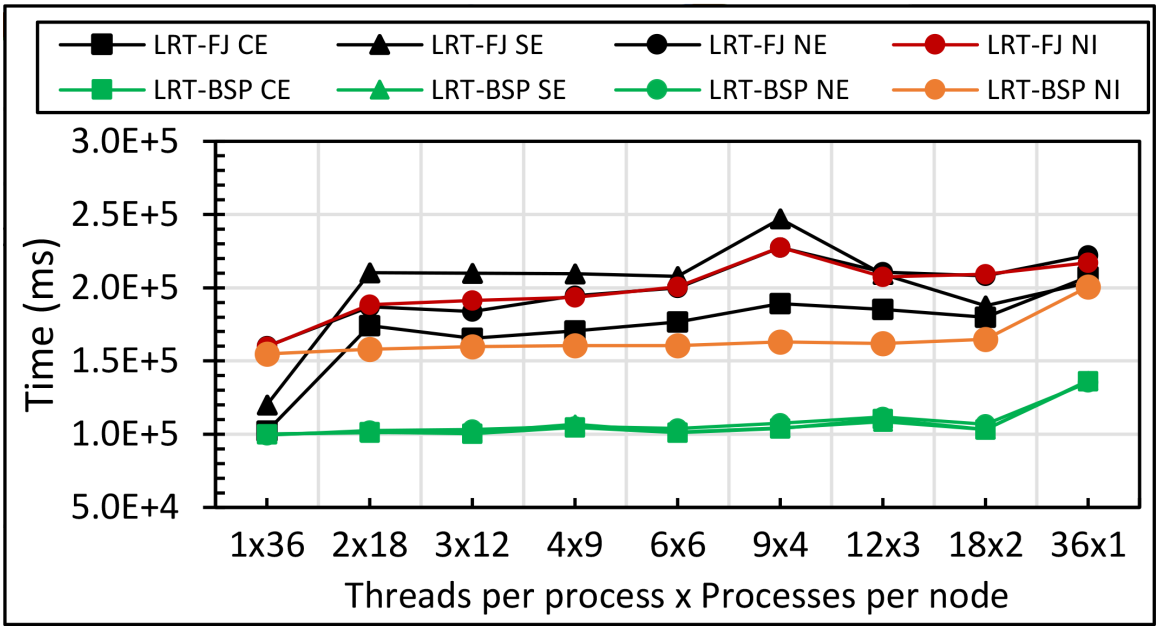


FIGURE 6.13. Java DA-MDS 50k points performance on 16 of 36-core nodes for LRT-FJ and LRT-BSP over varying threads and processes. Affinity patterns are CE,NE,SE, and NI.

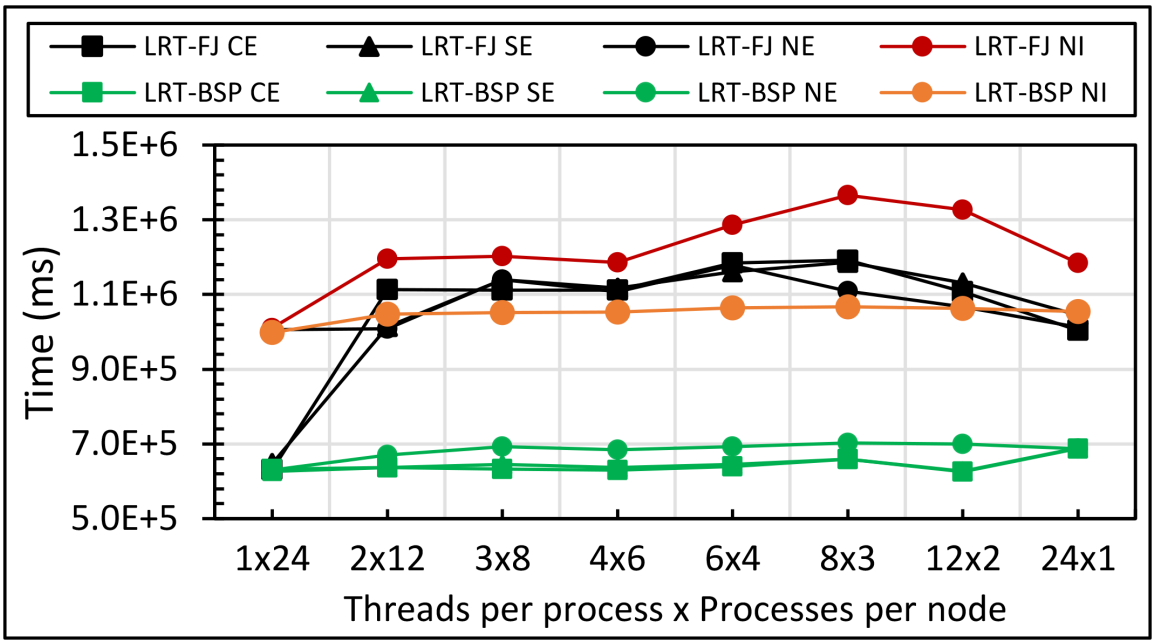


FIGURE 6.14. Java DA-MDS 100k points performance on 16 nodes for LRT-FJ and LRT-BSP over varying threads and processes. Affinity patterns are CE,NE,SE, and NI.

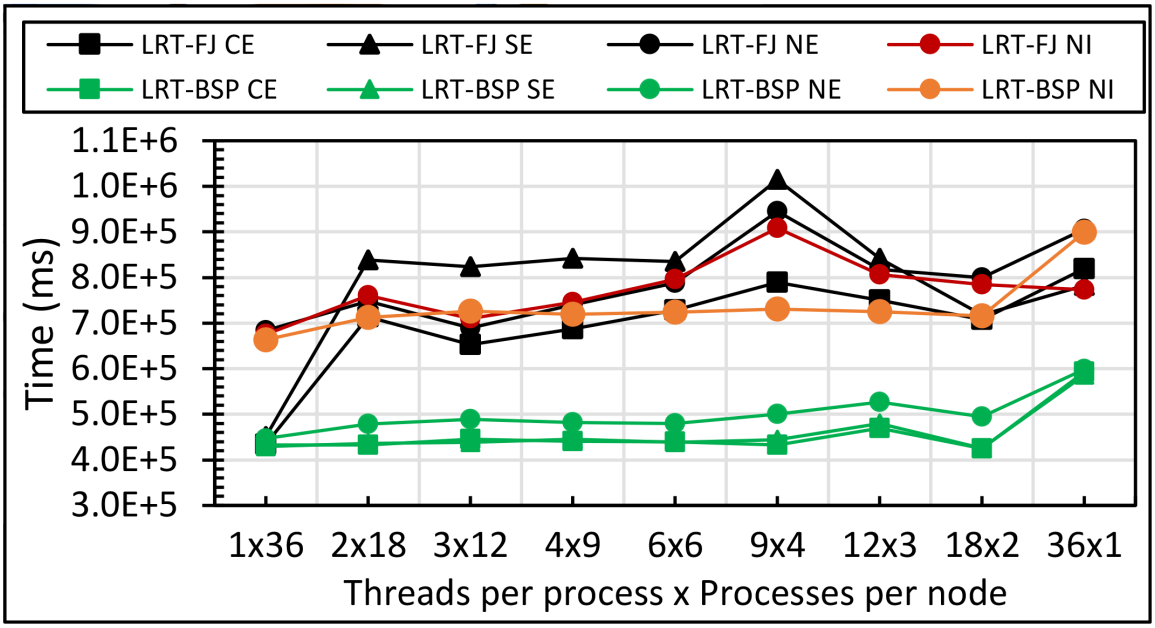


FIGURE 6.15. Java DA-MDS 100k points performance on 16 of 36-core nodes for LRT-FJ and LRT-BSP over varying threads and processes. Affinity patterns are CE,NE,SE, and NI.

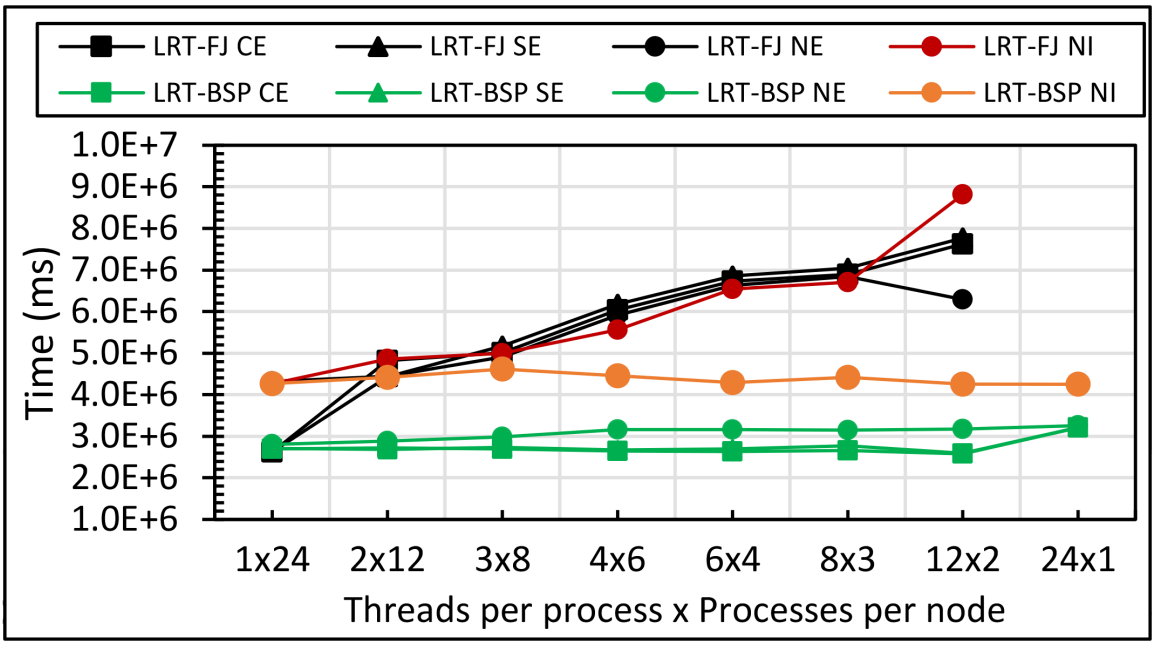


FIGURE 6.16. Java DA-MDS 200k points performance on 16 nodes for LRT-FJ and LRT-BSP over varying threads and processes. Affinity patterns are CE,NE,SE, and NI.

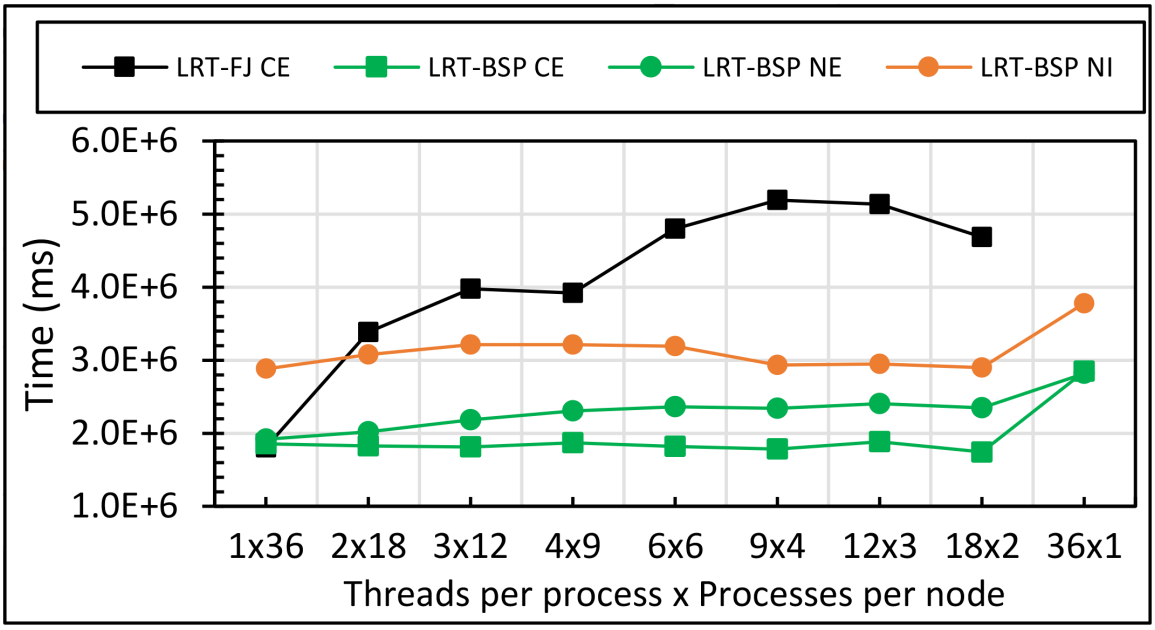


FIGURE 6.17. Java DA-MDS 200k points performance on 16 of 36-core nodes for LRT-FJ and LRT-BSP over varying threads and processes. Affinity patterns are CE,NE,SE, and NI.

TABLE 6.2. Java DA-MDS speedup for varying data sizes on 24-core and 36-core nodes. Red values indicate the suboptimal performance of LRT-FJ model compared to LRT-BSP. Ideally, these values should be similar to their immediate left cell values.

	Data Size								
	50k			**100k**			**200k**		
24-Core Nodes	**1x24 LRT-BSP**	**12x2 LRT-BSP**	**12x2 LRT-FJ**	**1x24 LRT-BSP**	**12x2 LRT-BSP**	**12x2 LRT-FJ**	**1x24 LRT-BSP**	**12x2 LRT-BSP**	**12x2 LRT-FJ**
16	1	1	0.6	1	1	0.6	1	1	0.4
32	2.2	2	1.1	1.9	1.9	1.1	1.9	2	0.6
64	3.9	3.6	1.9	3.6	3.6	1.9	3.7	3.8	0.9
36-Core Nodes	**1x36 LRT-BSP**	**18x2 LRT-BSP**	**18x2 LRT-FJ**	**1x36 LRT-BSP**	**18x2 LRT-BSP**	**18x2 LRT-FJ**	**1x36 LRT-BSP**	**18x2 LRT-BSP**	**18x2 LRT-FJ**
16	1	1	0.6	1	1	0.6	1	1.1	0.4
32	2	1.8	0.9	1.9	1.9	1.1	1.9	2.1	0.6

times that of the corresponding 200k run, so to be within our HPC resource allocation limits we have limited the number of iterations in 400k tests to a lesser value than those of 200k runs. Note that this does not affect performance characteristics in any way, as each iteration is independent and the number of iterations determines only the accuracy of results.

The red line in Figure 6.18 and Figure 6.19 show the LRT-FJ implementation using standard OpenMPI collective calls. It does not include any other optimizations discussed in the Section 5. Both the 100k and 200k results exhibit a high degree of variation in performance across the different thread and process combinations. Notably, the all-process, 1x24 pattern is worse in both cases due to the cost of communication. Also, the standard implementation has no GC or memory optimizations, which made it impossible to test the 400k data set, as the memory requirement exceeded the physical 128GB memory in Juliet. The blue line in these graphs shows the performance improvement when communication is done through shared memory and the thread model is LRT-FJ. It also includes static memory allocation, which made it possible to run 400k data without hitting the memory limit. The black line adds other optimizations that give extra performance improvements beyond mere shared memory communication. These optimizations are collectively identified as SPIDAL Java in the following graphs. The green line in the figures is SPIDAL Java with thread model changed to LRT-BSP, which gives the best performance.

Figure 6.21 and Figure 6.22 look at communication times of 100k and 200k tests. DA-MDS has two call sites to `MPI_Allgatherv`, identified by `BCComm` and `MMComm` in these graphs. The figures clearly show the poor performance of non-shared-memory-based communication in standard MPI. With the memory-map-based optimization discussed in

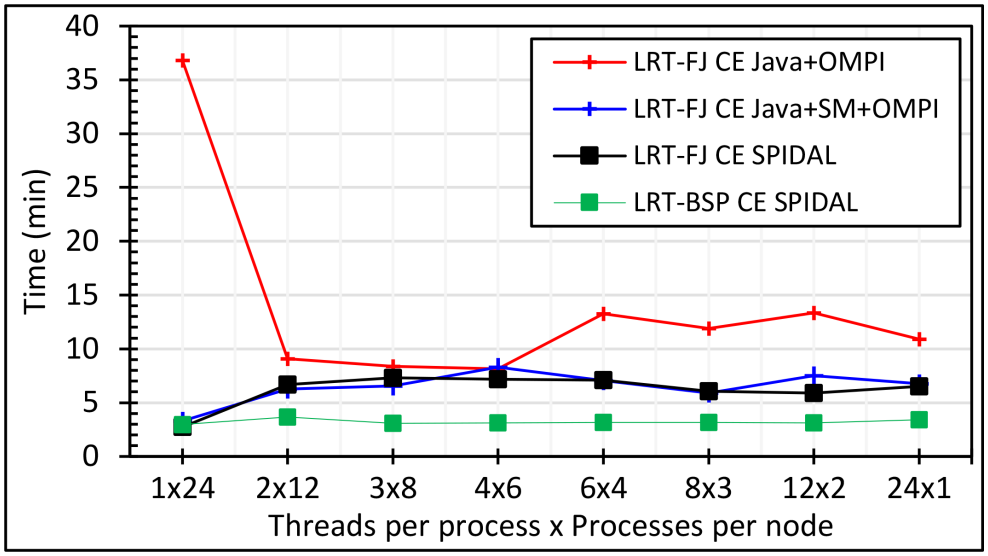


FIGURE 6.18. DA-MDS 100K performance with varying intra-node parallelism

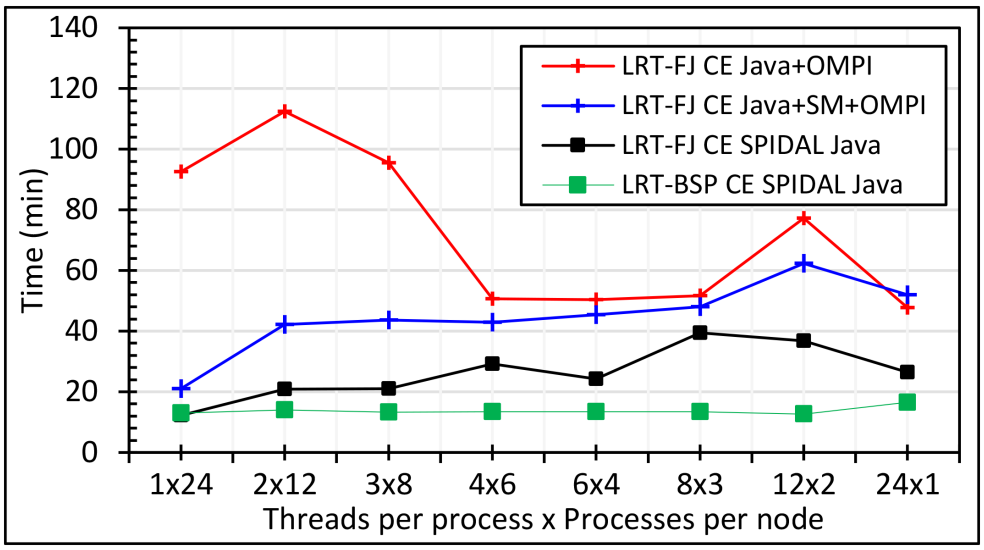


FIGURE 6.19. DA-MDS 200K performance with varying intra-node parallelism

Section 5, the cost of collective calls is constant across different patterns. Also, the communication costs follow linear growth with data size, as seen in the 1ms to 2ms increase when going from 100k to 200k.

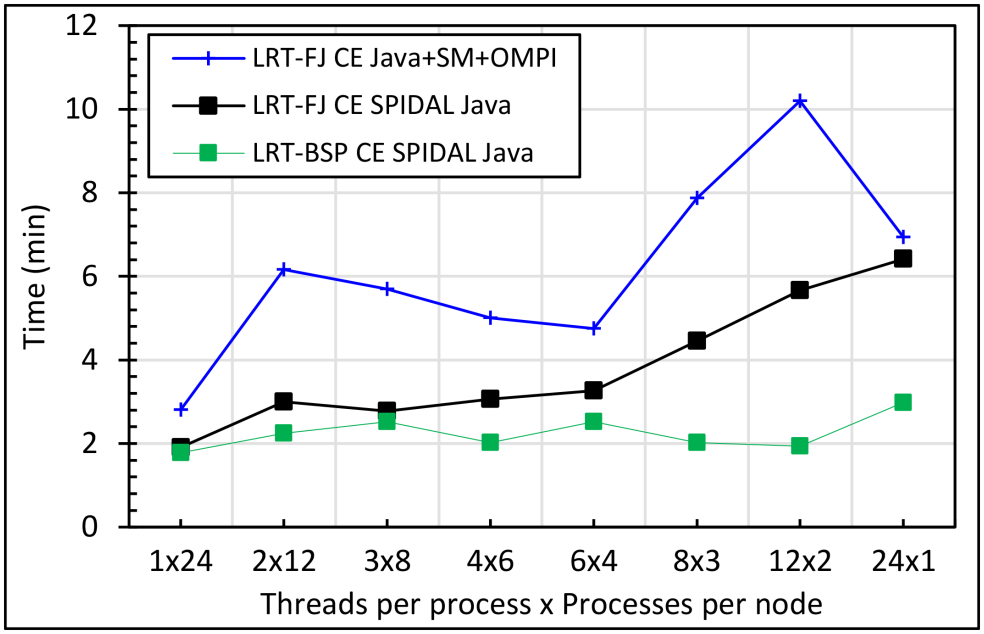


FIGURE 6.20. DA-MDS 400K performance with varying intra-node parallelism

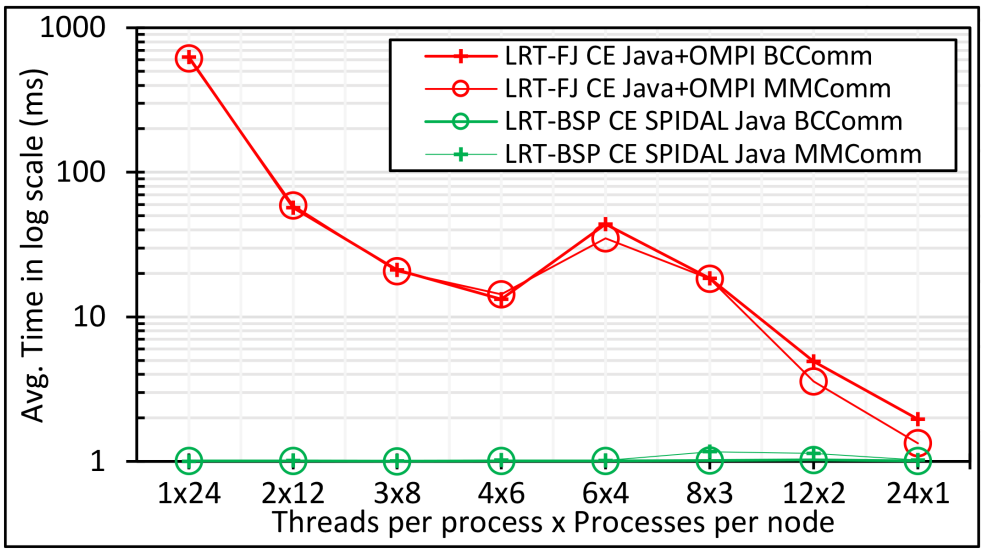


FIGURE 6.21. DA-MDS 100K `allgatherv` performance with varying intra-node parallelism

Figure 6.23 shows DA-MDS speedup with different optimization techniques for 200K data. Parallelism up to and including 1152 was achieved by increasing intra-node parallelism over 48 nodes. For example, 1152-way parallelism is 24-way internal across 48

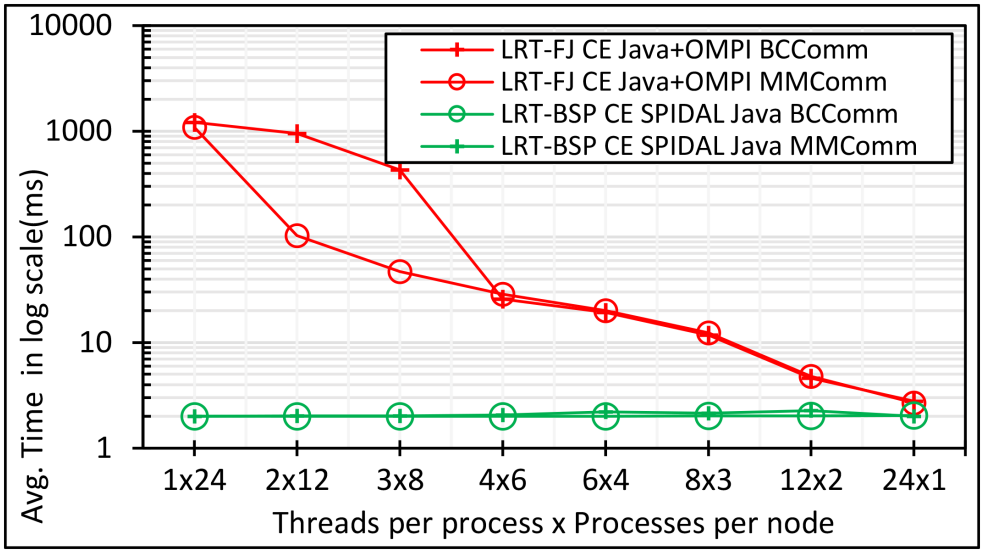


FIGURE 6.22. DA-MDS 200K `allgatherv` performance with varying intra-node parallelism

nodes. Parallelisms beyond 1152 were 24-way internal with the number of nodes alternating between 96 and 128. The top green line denotes SPIDAL Java optimizations and all processes. The purple line overlying the green is the SPIDAL Java LRT-BSP hybrid threads. Here, parallelisms up to 768 were using threads internally to a node and MPI across nodes. Parallelisms 768 through 3072 were two processes per node each running multiple threads internally over 48, 96, and 128 nodes. For example, 784-way parallelism was 8x2x48 and 1152-way was 12x2x48. The hybrid LRT-BSP threads and all-processes lines gave the best speedup. The next best case is LRT-BSP all-threads line depicted in orange color. It uses threads internally to a node, so parallelisms from 1,152 and upwards have 24 threads internally. From the previous results it is clear that using 24 threads is not efficient, as Juliet nodes have two physical CPUs. This explains the performance degradation compared to the hybrid threads case.

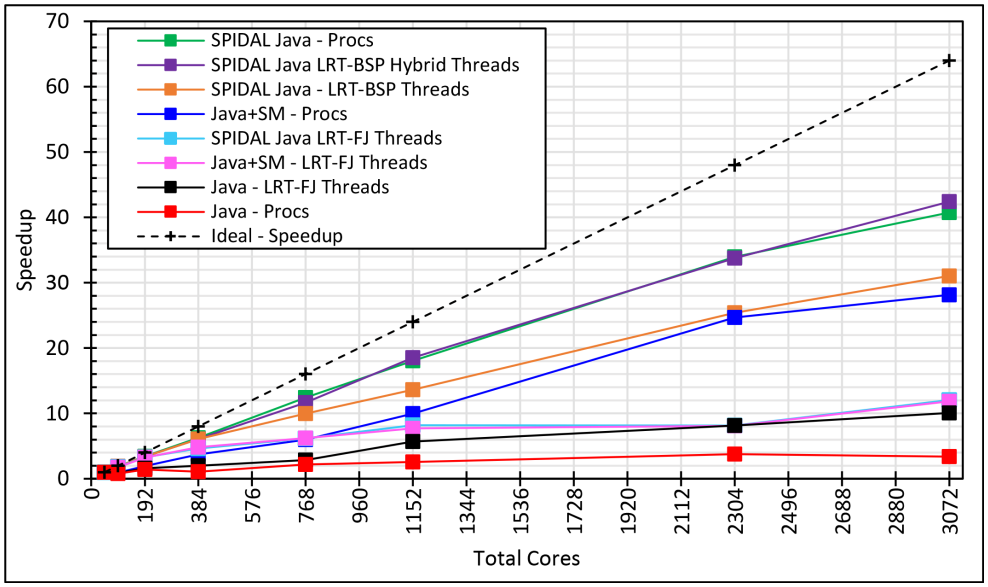


FIGURE 6.23. DA-MDS speedup for 200K with different optimization techniques

The blue line in Figure 6.23 is the all-processes case, which is similar to the green line except that blue only represents communication and memory optimizations. The light blue line shows the case with all the SPIDAL optimizations and using LRT-FJ threads internally. Comparing this to the orange line, it is evident how LRT-BSP improves performance significantly even when other optimizations are kept the same. The pink line has the same optimizations as the earlier dark blue line (Java+SM - Procs) but uses LRT-FJ threads. Both the pink and light blue lines overlap, suggesting that the overhead of LRT-FJ model is too high, making other optimizations insignificant. The last two lines are the standard Java plus MPI and thread implementations. The red line represents all processes, and the black line represents all threads internally and using LRT-FJ model. With the overhead from typical MPI collective communications, the all-processes (red) line shows much slower performance compared to even the LRT-FJ all-threads (black) line.

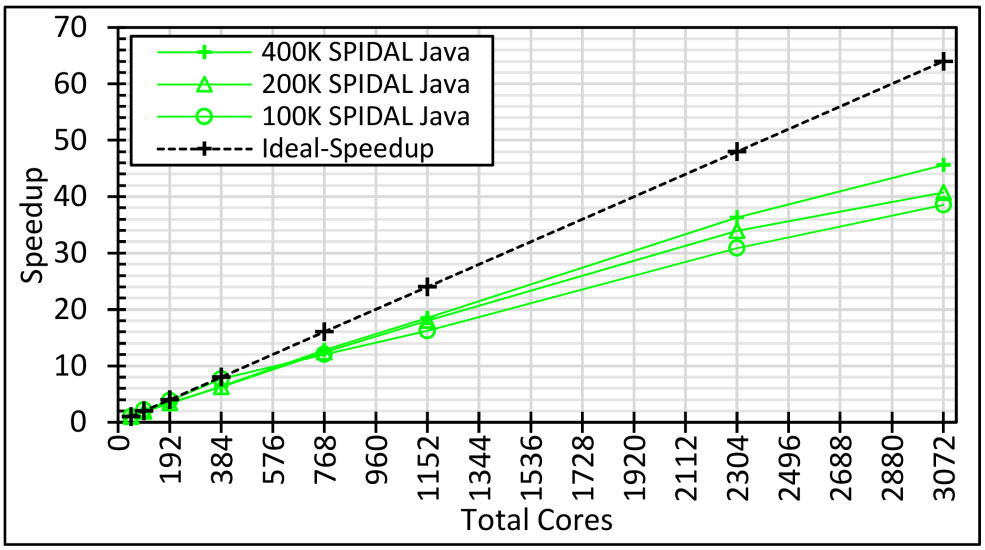


FIGURE 6.24. DA-MDS speedup with varying data sizes

Figure 6.24 extends the SPIDAL Java all-processes line in Figure 6.23 to other data sizes. With computations growing at $O(N^2)$ and communications at $O(N)$, better speedups should be achieved as data size increases. The three lines in the figure confirm this expectation, showing the highest speedup for 400k data.

6.3. Performance of MDSasChisq and DA-PWC

MDSasChisq is similar to DA-MDS in functionality but is based on the LevenbergMarquardt algorithm [65]. it does not support optimizations included in DA-MDS currently. Similarly, DA-PWC supports only a limited number of optimizations. The following results show that the performance of these algorithms in the absence of any optimizations agrees with that of DA-MDS and K-Means implying MDSasChisq and DA-PWC could benefit from the same optimizations such as SM based communications, LRT-BSP threads, and correct thread pinning.

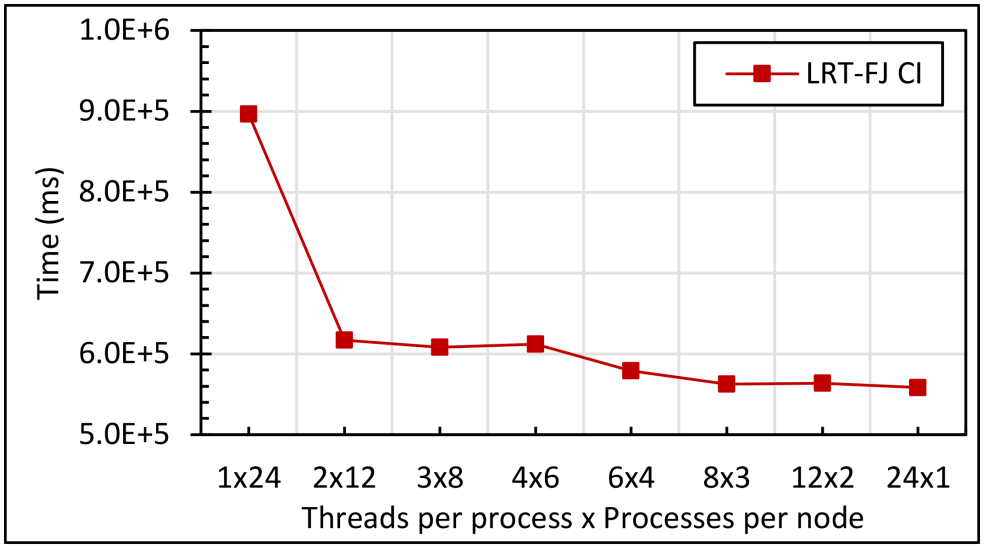


FIGURE 6.25. Java MDSasChisq 10k points performance on 32 nodes for LRT-FJ over varying threads and processes. Affinity pattern is `CI`

Figure 6.25 illustrates MDSasChisq LRT-FJ performance over varying threads and processes combinations. The poor performance of all-process – 1x24 – case suggest MDSasChisq could benefit from SM based communications discussed previously.

Figure 6.26 shows the effect of varying intra-node parallelism for all-process and all-threads approaches. With no SM based communication in all-process case, all-threads show better speedup. However, both these cases deviate greatly from the ideal speedup line as the number parallel tasks increase beyond 4 in this case.

Figure 6.27 illustrates DA-PWC LRT-FJ performance over varying threads and processes combinations. It supports SM based communication, which explains the better performance for 1x24 case over other patterns. Affinity and LRT-BSP threads could further improve DA-PWC.

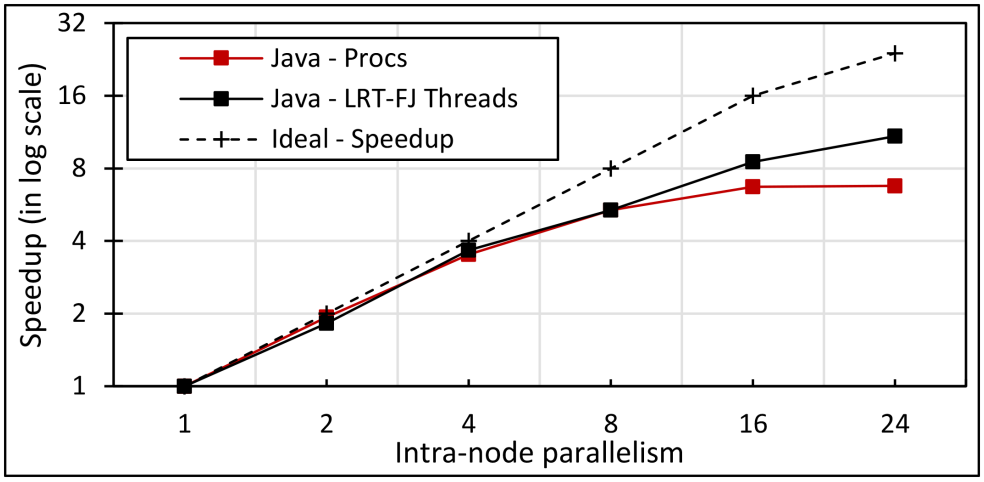


FIGURE 6.26. Java MDSasChisq 10k points speedup on 32 nodes for LRT-FJ over varying intra-node parallelism. Affinity pattern is `CI`

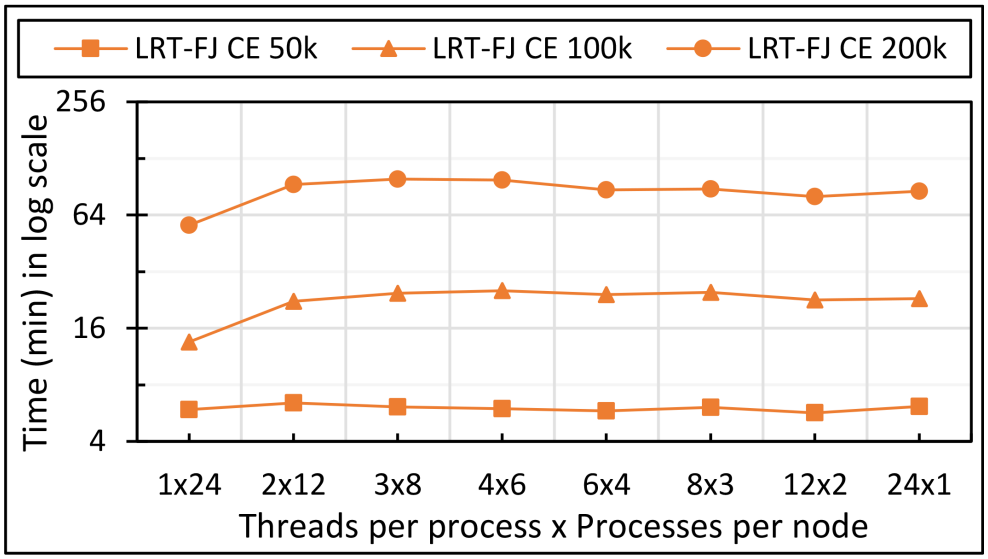


FIGURE 6.27. Java DA-PWC LRT-FJ performance on 32 nodes over varying varying threads and processes. Affinity pattern is `CE`

CHAPTER 7

Conclusion

While there are frameworks to support complex Big Data analytics, programming them for high performance is challenging. This difficulty is partly due to running them on virtualized commodity infrastructure, but we have found performance gains to be insignificant even on advanced HPC environments.

We have performed a systematic study of performance scaling across nodes and internally over many cores using two parallel machine-learning applications in Chapter 6. This study identified that the nature of Big Data problems and the use of high-level languages such as Java incur significant computation and communication costs, which hinder performance scaling. Chapter 5 introduced these factors in detail, where the thread model, affinity, and the communication mechanism are prominent factors.

The typical thread model used in parallel applications is the LRT-FJ model, which causes threads to release and acquire CPU resources frequently. The implementations of this model behave poorly and suffer greatly from context switches. The LRT-BSP model introduced in Chapter 5 overcomes these overheads by keeping threads busy throughout the application. The affinity of processes and threads matter as much as the thread model, where unbounded threads hinder performance even with LRT-BSP. The best affinity strategy was to pin worker threads to a separate core while setting the process affinity to all the cores utilized by threads (affinity pattern `CE`). Also, it was revealed that threads should be kept

within NUMA boundaries to avoid memory access costs across physical CPUs. The results of both DA-MDS and K-Means supported these findings and exemplified that the hybrid pattern of one process per socket and all-threads internally to a socket produced the best results over any other pattern having threads internally to a process. While thread pinning is discussed in the literature, this research is the first to perform a detailed analysis over six strategies, especially for Java threads. The third improvement introduced in Chapter 5 is the shared-memory-based communication for processes within the same node. The typical inter-process communication proved prohibitively expensive, especially when the number of processes per node grows as large as there are cores. The memory-mapping technique introduced in this dissertation allowed Java processes to communicate with a minimal and constant cost despite the number of processes. Apart from the Chronicle Queue [**63**] software, this work is the first to use memory mapping with Java to do inter-process communication for Big Data.

Chapter 6 also looked at the performance of two Big Data systems, Apache Spark and Flink that implement the dataflow model. It suggested that while the dataflow programming model is attractive, its implementations can be further improved based on the previous findings of DA-MDS and K-Means clustering.

The work presented in this dissertation shows that Java Big Data performance could be improved significantly by adhering to certain advanced techniques. While it focuses on parallel machine-learning applications, the experience and guidelines can be generalized in performing systematic performance studies over other Big Data applications and frameworks. For example, bringing systems such as Apache Spark and Flink into HPC

while retaining the productivity and usability features but improving performance based on the ideas presented here would constitute a promising future research direction of this work. Another step would be to extend and apply the CD classification in Chapter 4 for existing Big Data use cases. Finally, an indirect research direction would be to come up with high-performance idioms to write Big Data applications on existing frameworks.

Bibliography

[1] Apache Flink: Scalable Batch and Stream Data Processing.

[2] *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.

[3] I. Anagnostopoulos, S. Zeadally, and E. Exposito. Handling big data: research challenges and future directions. *The Journal of Supercomputing*, 72(4):1494–1516, 2016.

[4] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[5] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph, 2013.

[6] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[7] Mark Baker, Bryan Carpenter, and Aamir Shafi. MPJ Express: towards thread safe Java HPC. In *2006 IEEE International Conference on Cluster Computing*, pages 1–10. IEEE, 2006.

[8] Chaitanya Baru, Milind Bhandarkar, Raghunath Nambiar, Meikel Poess, and Tilmann Rabl. *Setting the Direction for Big Data Benchmark Standards*, pages 197–208. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[9] L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. Scalapack: A portable linear algebra library for distributed

memory computers - design issues and performance. In *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pages 5–5, 1996.

[10] I. Borg and P.J.F. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer, 2005.

[11] David Camp, Christoph Garth, Hank Childs, David Pugmire, and Kenneth Joy. Streamline integration using MPI-hybrid parallelism on a large multicore architecture. *IEEE Transactions on Visualization and Computer Graphics*, 17(11):1702–1713, 2011.

[12] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.

[13] Maria Carpen-Amarie, Patrick Marlier, Pascal Felber, and Gaël Thomas. A performance study of java garbage collectors on multicore architectures. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 20–29. ACM, 2015.

[14] Martin J Chorley and David W Walker. Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters. *Journal of Computational Science*, 1(3):168–174, 2010.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[16] Microsoft Corporation. Microsoft mpi. https://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx, 2016. Accessed: May 10 2016.

[17] Oracle Corporation. Java i/o streams. https://docs.oracle.com/javase/tutorial/essential/io/streams.html, 2015. Accessed: September 8 2016.

[18] Oracle Corporation. Java direct buffers. https://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html, 2016. Accessed: September 8 2016.

[19] Standard Performance Evaluation Corporation. Spec cpu 2006. https://www.spec.org/cpu2006/, 2015. Accessed: September 8 2016.

[20] Transaction Processing Council. Tpc express benchmark hs specification. http://www.tpc.org/tpc_documents_current_versions/pdf/tpcx-hs_v1.4.1.pdf, 2016. Accessed: September 8 2016.

[21] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[23] Jack J. Dongarra. *Supercomputing: 1st International Conference Athens, Greece, June 8–12, 1987 Proceedings*, chapter The LINPACK Benchmark: An explanation, pages 456–474. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988.

[24] Jack J. Dongarra. Performance of various computers using standard linear equations software. *SIGARCH Comput. Archit. News*, 20(3):22–44, June 1992.

[25] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: Past, present, and future. concurrency and computation: Practice and experience. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[26] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A Runtime for Iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[27] Saliya Ekanayake. Study of biological sequence structure: Clustering and visualization. http://grids.ucs.indiana.edu/ptliupages/publications/study_of_sequence_clustering_formatted_v2.pdf, 2013. Accessed: September 8 2016.

[28] Saliya Ekanayake. C mpi k-means. https://github.com/DSC-SPIDAL/KMeansC, 2016. Accessed: May 4, 2016.

[29] Saliya Ekanayake. Java mpi k-means. https://github.com/DSC-SPIDAL/KMeans, 2016. Accessed: May 4, 2016.

[30] Saliya Ekanayake, Supun Kamburugamuve, and Geoffrey Fox. Spidal: High performance data analytics with java and mpi on large multicore hpc clusters. In *Proceedings of the 2016 Spring Simulation Multi-Conference (SPRINGSIM)*, Pasadena, CA, USA, 3–6, 2016.

[31] Charles Elkan. Using the triangle inequality to accelerate k-means. In Tom Fawcett and Nina Mishra, editors, *ICML*, pages 147–153. AAAI Press, 2003.

[32] Roberto R. Expósito, Sabela Ramos, Guillermo L. Taboada, Juan Touriño, and Ramón Doallo. Fastmpj: A scalable and efficient java message-passing library. *Cluster Computing*, 17(3):1031–1050, September 2014.

[33] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

[34] Apache Software Foundation. Terasort. https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/examples/terasort/package-summary.html, 2015. Accessed: September 8 2016.

[35] G. Fox, D.R. Mani, and S. Pyne. Parallel deterministic annealing clustering and its application to lc-ms data analysis. In *Big Data, 2013 IEEE International Conference on*, pages 665–673, Oct 2013.

[36] G.C. Fox, J. Qiu, S. Kamburugamuve, S. Jha, and A. Luckow. Hpc-abds high performance computing enhanced apache big data stack. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 1057–1066, May 2015.

[37] Geoffrey Fox. Robust scalable visualized clustering in vector and non vector semi-metric spaces. *Parallel Processing Letters*, 23(2), 2013.

[38] Geoffrey Fox, Judy Qiu, Shantenu Jha, Saliya Ekanayake, and Supun Kamburugamuve. Big data, simulations and hpc convergence. http://dsc.soic.indiana.edu/publications/HPCBigDataConvergence.pdf, 2016. Accessed: June 16 2016.

[39] Geoffrey C. Fox. Deterministic annealing and robust scalable data mining for the data deluge. In *Proceedings of the 2Nd International Workshop on Petascal Data Analytics: Challenges and Opportunities*, PDAC '11, pages 39–40, New York, NY, USA, 2011. ACM.

[40] Geoffrey C. Fox, Shantenu Jha, Judy Qiu, and Andre Luckow. Towards an Understanding of Facets and Exemplars of Big Data Applications. In *Proceedings of the 20 Years of Beowulf Workshop on Honor of Thomas Sterling's 65th Birthday*, Beowulf '14, pages 7–16, New York, NY, USA, 2015. ACM.

[41] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1197–1208, New York, NY, USA, 2013. ACM.

[42] Bhaskar D Gowda and Nishkam Ravi. Bigbench: Toward an industry-standard benchmark for big data analytics. http://blog.cloudera.com/blog/2014/11/bigbench-toward-an-industry-standard-benchmark-for-big-data-analytics/, 2014. Accessed: Ja 04 2015.

[43] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. *Open MPI: A Flexible High Performance MPI*, pages 228–239. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[44] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-passing Interface*. MIT Press, Cambridge, MA, USA, 1994.

[45] Munish Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

[46] Geoffrey L. House, Saliya Ekanayake, Yang Ruan, Ursel M.E. Schtte, Wittaya Kaonongbua, Geoffrey Fox, Yuzhen Ye, and James D. Bever. Phylogenetically structured differences in rrna gene sequence variation among species of arbuscular mycorrhizal fungi and their implications for sequence clustering. *Applied and Environmental Microbiology*, 2016.

[47] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. *The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis*, volume 74 of *Lecture Notes in Business Information Processing*, book section 9, pages 209–228. Springer Berlin Heidelberg, 2011.

[48] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. x. D. K. Panda. Design of high performance mvapich2: Mpi2 over infiniband. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, pages 43–48, Washington, DC, USA, 2006. IEEE Computer Society.

[49] W. Huang, G. Santhanaraman, H.W. Jin, Q. Gao, and D.K. Panda. Design of high performance mvapich2: Mpi2 over infiniband. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 43–48, May 2006.

[50] Shams Imam and Vivek Sarkar. Habanero-java library: A java 8 framework for multicore programming. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 75–86, New York, NY, USA, 2014. ACM.

[51] Cray Inc. Cray message passing toolkit (mpt). http://docs.cray.com/cgi-bin/craydoc.cgi?mode=View;id=sw_releases-o23alcrv-1426185385;idx=man_search;this_sort=title;q=;type=man;title=Message%20Passing%20Toolkit%20%28MPT%29%207.2%20Man%20Pages, 2016. Accessed: May 10 2016.

[52] Intel. Intel mpi library. https://software.intel.com/en-us/intel-mpi-library, 2016. Accessed: May 10 2016.

[53] Zhen Jia, Jianfeng Zhan, Lei Wang, Rui Han, Sally A. McKee, Qiang Yang, Chunjie Luo, and Jingwei Li. Characterizing and subsetting big data workloads. *CoRR*, abs/1409.0792, 2014.

[54] Supun Kamburugamuve. Flink k-means. https://github.com/DSC-SPIDAL/flink-apps, 2016. Accessed: May 4, 2016.

[55] Supun Kamburugamuve, Saliya Ekanayake, Milinda Pathirage, and Geoffrey Fox. Towards High Performance Processing of Streaming Data in Large Data Centers. In *HPBDC 2016 IEEE International Workshop on High-Performance Big Data Computing in conjunction with The 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2016), Chicago, Illinois USA*, 2016.

[56] Supun Kamburugamuve, Pulasthi Wickramasinghe, Saliya Ekanayake, Milinda Pathirage, and Geoffrey C. Fox. Webplotviz. https://spidal-gw.dsc.soic.indiana.edu/, 2015. Accessed: September 8 2016.

[57] Supun Kamburugamuve, Pulasthi Wickramasinghe, Saliya Ekanayake, Chathuri Wimalasena, Milinda Pathirage, and Geoffrey Fox. Tsmap3d: Browser visualization of high dimensional time series data. http://dsc.soic.indiana.edu/publications/tsmap3d.pdf, 2016. Accessed: June 16 2016.

[58] Rajesh K Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009.

[59] Michael Klemm, Matthias Bezold, Ronald Veldema, and Michael Philippsen. JaMP: an implementation of OpenMP for a Java DSM. *Concurrency and Computation: Practice and Experience*, 19(18):2333–2352, 2007.

[60] Hansjörg Klock and Joachim M. Buhmann. Multidimensional scaling by deterministic annealing. In *Proceedings of the First International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition*, EMMCVPR '97, pages 245–260, London, UK, UK, 1997. Springer-Verlag.

[61] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[62] Berkeley AMP Labs. Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/, 2014. Accessed: September 8 2016.

[63] Peter Lawrey. Chronicle queue. http://chronicle.software/products/chronicle-queue/, 2015. Accessed: June 16 2016.

[64] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[65] Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly Journal of Applied Mathmatics*, II(2):164–168, 1944.

[66] Chunjie Luo, Wanling Gao, Zhen Jia, Rui Han, Jingwei Li, Xinlong Lin, Lei Wang, Yuqing Zhu, and Jianfeng Zhan. Handbook of bigdatabench (version 3.1) - a big data benchmark suite. Report.

[67] D. A. Malln, G. L. Taboada, J. Tourio, and R. Doallo. Npb-mpj: Nas parallel benchmarks implementation for message-passing in java. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 181–190, Feb 2009.

[68] Big data: The next frontier for innovation, competition, and productivity. http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation.

[69] Xiangrui Meng, Joseph Bradley, B Yuvaz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(34):1–7, 2016.

[70] Ramanathan Narayanan, Berkin O zs.kylmaz, Joseph Zambreno, Gokhan Memik, Alok Choudhary, and Jayaprakash Pisharath. Minebench: A benchmark suite for data mining workloads. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 182–188.

[71] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970.

[72] OpenHFT JavaLang Project. https://github.com/OpenHFT/Java-Lang.

[73] Jeffrey M. Squyres Oscar Vega-Gisbert, Jose E. Roman. Design and implementation of java bindings in open mpi. users.dsic.upv.es/~jroman/preprints/ompi-java.pdf.

[74] OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[75] B. Ozisikyilmaz, R. Narayanan, J. Zambreno, G. Memik, and A. Choudhary. An architectural characterization study of data mining and bioinformatics workloads. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 61–70.

[76] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel Distrib. Technol.*, 4(2):63–79, June 1996.

[77] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *2009 17th Euromicro international conference on parallel, distributed and network-based processing*, pages 427–436. IEEE, 2009.

[78] Tilmann Rabl and Meikel Poess. Parallel data generation for performance analysis of large, complex rdbms. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, DBTest '11, pages 5:1–5:6, New York, NY, USA, 2011. ACM.

[79] Tilmann Rabl, Meikel Poess, Chaitanya Baru, and Hans-Arno Jacobsen. Specifying big data benchmarks: First workshop, wbdb 2012, san jose, ca, usa, may 8-9, 2012 and second workshop, wbdb 2012, pune, india, december 17-18, 2012, revised selected papers. Springer Berlin Heidelberg.

[80] Karthik Ramasamy. Flying faster with twitter heron. https://blog.twitter.com/2015/flying-faster-with-twitter-heron, 2015. Accessed: September 8 2016.

[81] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[82] K. Rose, E. Gurewwitz, and G. Fox. A deterministic annealing approach to clustering. *Pattern Recogn. Lett.*, 11(9):589–594, September 1990.

[83] Yang Ruan, Saliya Ekanayake, Mina Rho, Haixu Tang, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Dacidr: Deterministic annealed clustering with interpolative dimension reduction using a large collection of 16s rrna sequences. In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, BCB '12, pages 329–336, New York, NY, USA, 2012. ACM.

[84] Yang Ruan and Geoffrey Fox. A robust and scalable solution for interpolative multidimensional scaling with weighting. In *9th IEEE International Conference on eScience, eScience 2013, Beijing, China, October 22-25, 2013*, pages 61–69, 2013.

[85] Yang Ruan, Geoffrey L. House, Saliya Ekanayake, Ursel Schutte, James D. Bever, Haixu Tang, and Geoffrey C. Fox. Integration of clustering and multidimensional scaling to determine phylogenetic trees as spherical phylograms visualized in 3 dimensions. In *CCGRID*, pages 720–729. IEEE Computer Society, 2014.

[86] Eishay Smith. Jvm serializers. https://github.com/eishay/jvm-serializers/wiki, 2016. Accessed: September 8 2016.

[87] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[88] Esoteric Software. Kryo. https://github.com/EsotericSoftware/kryo, 2016. Accessed: May 10 2016.

[89] Larissa Stanberry, Roger Higdon, Winston Haynes, Natali Kolker, William Broomall, Saliya Ekanayake, Adam Hughes, Yang Ruan, Judy Qiu, Eugene Kolker, and Geoffrey Fox. Visualizing the protein sequence universe. In *Proceedings of the 3rd International Workshop on Emerging Computational Methods for the Life Sciences*, ECMLS '12, pages 13–22, New York, NY, USA, 2012. ACM.

[90] Guillermo L Taboada, Sabela Ramos, Roberto R Expósito, Juan Touriño, and Ramón Doallo. Java in the high performance computing arena: Research, practice and experience. *Science of Computer Programming*, 78(5):425–444, 2013.

[91] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[92] Oscar Vega-Gisbert, Jose E. Roman, Siegmar Groß, and Jeffrey M. Squyres. Towards the availability of java bindings in open mpi. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 141–142, New York, NY, USA, 2013. ACM.

[93] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499, Feb 2014.

[94] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. Bigdatabench: a big data benchmark suite from internet services. *CoRR*, abs/1401.1406, 2014.

[95] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[96] Reynold Xin and Josh Rosen. Project tungsten: Bringing apache spark closer to bare metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html, 2015. Accessed: June 6, 2016.

[97] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[98] 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*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[99] B. Zhang, Y. Ruan, and J. Qiu. Harp: Collective communication on hadoop. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 228–233, March 2015.

[100] Bingjing Zhang, Yang Ruan, and Judy Qiu. Harp: Collective communication on hadoop. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 228–233. IEEE, 2015.

Saliya Ekanayake

Research Interests

Developing Big Data applications and systems, distributed systems, parallel machine learning, high-performance computing, parallel architectures.

Education

Indiana University
Ph.D., Computer Science, October 2016.

Indiana University
M.Sc., Computer Science, May 2011.

University of Moratuwa, Sri Lanka
B.Sc., Computer Science and Engineering, May 2008.

Employment Experience

Indiana University, Research Assistant, August 2009 – October 2016.

WSO2 Inc., Senior Software Engineer, January 2009 – August 2009.

WSO2 Inc., Software Engineer, May 2008 – January 2009.

University of Moratuwa, Sri Lanka, Visiting Lecturer, May 2008 – August 2009.

St. Mary's Convent, Sri Lanka, Assistant Physics Teacher, May 2003 – January 2004.

Publications

[1] **Saliya Ekanayake**, S. Kamburugamuve, P. Wickramasinghe, and G. C. Fox, "Java thread and process performance for parallel machine learning on multicore hpc clusters," *Submitted to IEEE Big Data*, 2016.

[2] G. L. House, **Saliya Ekanayake**, Y. Ruan, U. M. Schütte, W. Kaonongbua, G. Fox, Y. Ye, and J. D. Bever, "Phylogenetically structured differences in rrna gene sequence variation among species of arbuscular mycorrhizal fungi and their implications for sequence clustering," *Applied and Environmental Microbiology*, 2016.

[3] **Saliya Ekanayake**, S. Kamburugamuve, and G. C. Fox, "SPIDAL: High performance data analytics with java and mpi on large multicore hpc clusters," in *Proceedings of the 24th High Performance Computing Symposium (HPC 2016)*, Pasadena, California, 2016.

[4] S. Kamburugamuve, **Saliya Ekanayake**, M. Pathirage, and G. C. Fox, "Towards high performance processing of streaming data in large data centers," in *IPDPS Workshops*, IEEE Computer Society, 2016, pp. 1637–1644.

[5] G. C. Fox, S. Jha, J. Qiu, **Saliya Ekanayake**, and A. Luckow, "Towards a comprehensive set of big data benchmarks," in *High Performance Computing Workshop*, ser. Advances in Parallel Computing, vol. 26, IOS Press, 2014, pp. 47–66.

[6] Y. Ruan, G. L. House, **Saliya Ekanayake**, U. Schutte, J. D. Bever, H. Tang, and G. C. Fox, "Integration of clustering and multidimensional scaling to determine phylogenetic trees as spherical phylograms visualized in 3 dimensions," in *CCGRID*, IEEE Computer Society, 2014, pp. 720–729.

[7] Y. Ruan, **Saliya Ekanayake**, M. Rho, H. Tang, S.-H. Bae, J. Qiu, and G. Fox, "DACIDR: Deterministic annealed clustering with interpolative dimension reduction using a large collection of 16s rrna sequences," in *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, ser. BCB '12, Orlando, Florida: ACM, 2012, pp. 329–336.

[8] L. Stanberry, R. Higdon, W. Haynes, N. Kolker, W. Broomall, **Saliya Ekanayake**, A. Hughes, Y. Ruan, J. Qiu, E. Kolker, and G. C. Fox, "Visualizing the protein sequence universe," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 6, pp. 1313–1325, 2014.

[9] A. Hughes, Y. Ruan, **Saliya Ekanayake**, S. Bae, Q. Dong, M. Rho, J. Qiu, and G. C. Fox, "Interpolative multidimensional scaling techniques for the identification of clusters in very large sequence sets," *BMC Bioinformatics*, vol. 13, no. S-2, S9, 2012.

[10] J. Qiu, J. Ekanayake, T. Gunarathne, J. Y. Choi, S. Bae, H. Li, B. Zhang, T. Wu, Y. Ruan, **Saliya Ekanayake**, A. Hughes, and G. C. Fox, "Hybrid cloud and cluster computing paradigms for life science applications," *BMC Bioinformatics*, vol. 11, no. S-12, S3, 2010.

[11] J. Qiu, S. Beason, S. Bae, **Saliya Ekanayake**, and G. C. Fox, "Performance of windows multicore systems on threading and MPI," in *CCGRID*, IEEE Computer Society, 2010, pp. 814–819.

Awards and Honors

Best Intern Presentation Awarded by the Institute of Engineers in Sri Lanka (IESL) for implementing and presenting the JavaScript support in Apache Axis2 Web services engine.