Granules: A Lightweight, Streaming Runtime for Cloud Computing With Support for Map-Reduce

Shrideep Pallickara, Jaliya Ekanayake and Geoffrey Fox
Community Grids Laboratory, Indiana University
{spallick, jekanaya, gcf}@indiana.edu

Abstract

Cloud computing has gained significant traction in recent years. The Map-Reduce framework is currently the most dominant programming model in cloud computing settings. In this paper, we describe Granules, a lightweight, streaming-based runtime for cloud computing which incorporates support for the Map-Reduce framework. Granules provides rich lifecycle support for developing scientific applications with support for iterative, periodic and data driven semantics for individual computations and pipelines. We describe our support for variants of the Map-Reduce framework. The paper presents a survey of related work in this area. Finally, this paper describes our performance evaluation of various aspects of the system, including (where possible) comparisons with other comparable systems.

Keywords: map-reduce, cloud computing, streaming, cloud runtimes, and content distribution networks

1 Introduction

Cloud computing has gained significant traction in recent years. By facilitating access to an elastic (meaning the available resource pool can expand or contract over time) set of resources, cloud computing has demonstrable applicability to a wide-range of problems in several domains.

Appealing features within cloud computing include access to a vast number of computational resources and inherent resilience to failures. The latter feature arises, because in cloud computing the focus of execution is not a specific well-known resource but rather the best available one. Another characteristic of a lot of programs that have been written for cloud computing is that they tend to be stateless. Thus, when failures do take place, the appropriate computations are simply re-launched with the corresponding datasets.

Among the forces that have driven the need for cloud computing are falling hardware costs and burgeoning data volumes. The ability to procure cheaper, more powerful CPUs coupled with improvements in the quality and capacity of networks have made it possible to assemble clusters at increasingly attractive prices. The proliferation of networked devices, internet services, and simulations has resulted in large volumes of data being produced. This, in turn, has fueled the need to process and store vast amounts of data. These data volumes cannot be processed by a single computer or a small cluster of computer. Furthermore, in most cases, this data can be processed in a pleasingly parallel fashion. The result has been the aggregation of a large number of commodity hardware components in vast data centers.

Map-Reduce [1], introduced by Dean and Ghemawat at Google, is the most dominant programming model for developing applications in cloud settings. Here, large datasets are split into smaller more manageable sizes which are then processed by multiple *map* instances. The results produced by individual map functions are then sent to *reducers*, which collate these partial results to produce the final output. A clear benefit of such concurrent processing is a speed-up that is proportional to the number of computational resources. Map-Reduce can be thought of as an instance of the SPMD [2] programming model for parallel computing introduced by Federica Derema. Applications that can benefit from Map-Reduce include data and/or task-parallel algorithms in domains such as information retrieval, machine learning, graph theory and visualization among others.

In this paper we describe Granules [3], a lightweight streaming-based runtime for cloud computing. Granules allows processing tasks to be deployed on a single resource or a set of resources. Besides the basic support for Map-Reduce, we have incorporated support for variants of the Map-Reduce framework that are particularly suitable for scientific applications. Unlike most Map-Reduce implementations, Granules utilizes streaming for disseminating intermediate results, as opposed to using file-based communications. This leads to demonstrably better performance (see benchmarks in section 6).

This paper is organized as follows. In section 2, we provide a brief overview of the NaradaBrokering substrate that we use for streaming. We discuss some of the core elements of Granules in section 3. Section 4 outlines our support for Map-Reduce and for the creation of complex computational pipelines. In section 5, we describe related work in this area. In section 6, we profile several aspects of the Granules runtime, and where possible contrast its performance with comparable systems such as Hadoop, Dryad and MPI. In section 7, we present our conclusions.

2 NaradaBrokering

Granules uses the NaradaBrokering [4-6] streaming substrate (developed by us) for all its streams disseminations. The NaradaBrokering content distribution

network (depicted in Figure 1) comprises a set of cooperating router nodes known as *broker*s. Producers and consumers do not directly interact with each other. Entities, which are connected to one of the brokers within the broker network, use their hosting broker to funnel streams into the broker network and, from thereon, to other registered consumers of those streams.

NaradaBrokering is application-independent and incorporates several services to mitigate network-induced problems as streams traverse domains during disseminations. The system provisions easy to use guarantees, while delivering consistent and predictable performance that is adequate for use in real-time settings.

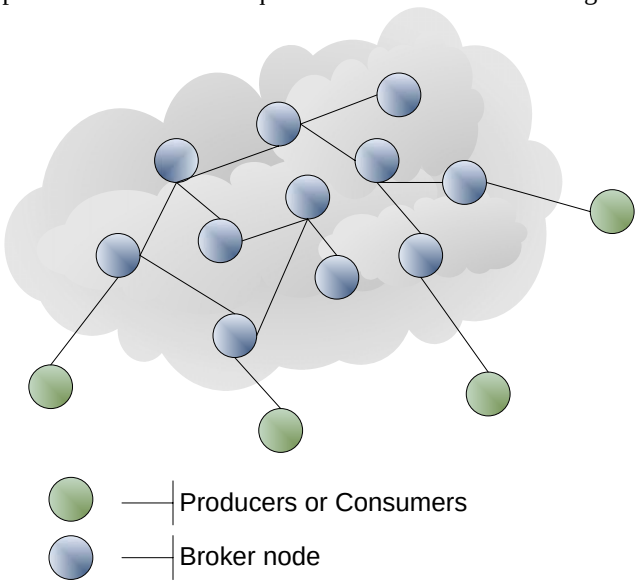

Figure 1: The NaradaBrokering broker network

Consumers of a given data stream can specify, very precisely, the portions of the data stream that they are interested in consuming. By preferentially deploying links during disseminations, the routing algorithm [4] in NaradaBrokering ensures that the underlying network is optimally utilized. This preferential routing ensures that consumers receive only those portions of streams that are of interest to them. Since a given consumer is typically interested in only a fraction of the streams present in the system, preferential routing ensures that a consumer is not deluged by streams that it will subsequently discard.

The system incorporates support for reliable streaming and secure streaming. In reliable streaming, the substrate copes with disconnects and process/link failures of different components within the system with the ability to fine-tune redundancies [5] for a specific stream. Secure streaming [6] enforces the authorization and confidentiality constraints associated with the generation and consumption of secure streams while coping with denial of service attacks.

Some of the domains that NaradaBrokering has been deployed in include earthquake science, particle physics, environmental monitoring, geosciences, GIS systems, and defense applications.

3 Granules

Granules orchestrates the concurrent execution of processing tasks on a distributed set of machines. Granules is itself distributed, and its components permeate not only the computational resources on which it interleaves processing, but also the desktop from where the applications are being deployed in the first place. The runtime manages the execution of a set of processing tasks through various stages of its lifecycle: deployment, initialization, execution and termination. Figure 2, depicts the various components that comprise Granules.

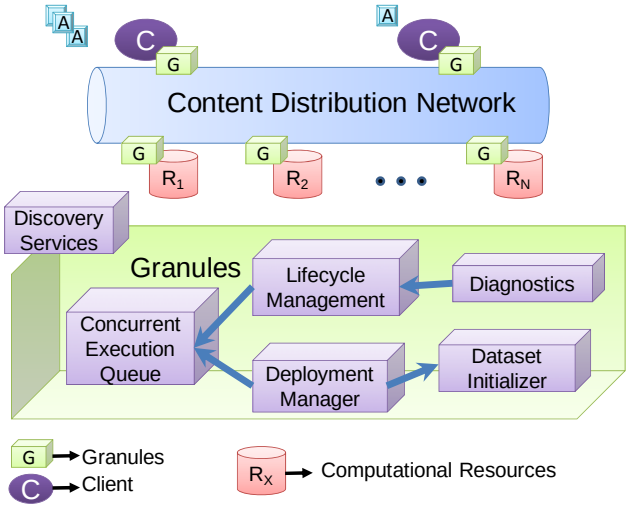

Figure 2: Overview of the Granules runtime

3.1 Computational task

The most fundamental unit in Granules is the notion of a *computational task*. This computational task encapsulates processing functionality, specifies its scheduling strategy and operates on different types of datasets. These computational tasks can take on additional interchangeable roles (such as map and reduce) and, when cascaded, can form complex execution pipelines.

Computational tasks require the domain specialists to specify processing functionality. This processing typically operate upon a collection of datasets encapsulated within the computational task.

The computational task encapsulates functionality for processing for a given fine grained unit of data. This data granularity could be a packet, a file, a set of files, or a database record. For example, a computational task can be written to evaluate a regular expression query (grep) on a set of characters, a file, or a set of files. In some cases there will not be a specific dataset; rather, each computational task instance initializes itself using a random-seed generator.

Computational tasks include several metadata such as versioning information, timestamps, domain identifiers and computation identifiers. Individual instances of the computational tasks include instance identifiers and task identifiers, which in turn allows us to group several related computational tasks together.

3.2 Datasets and collections

In Granules, datasets are used to simplify access to the underlying data type. Datasets currently supported within Granules include streams and files; support for databases is being incorporated. For a given data type, besides managing the allocation and reclamation of assorted resources, Granules also mediates access to it. For example, Granules performs actions related to simplifying the production and consumption of streams, the reading and writing of files, and transactional access to databases.

A data collection is associated with every computational task. The data collection represents a collection of datasets, and maintains information about the type, number and identifiers associated with every encapsulated dataset.

All that the domain specialist needs to specify is the number and type of the datasets involved. The system imposes no limits on the number of datasets within a dataset collection. During initializations of the dataset collection, depending on the type of the constituent datasets, Granules subscribes to the relevant streams, configures access to files on networked file systems, and sets up connections (JDBC) to the databases.

Dataset collections allow observers to be registered to track data availability, and dataset initializations and closure. This simplifies data processing since it obviates the need to perform polling.

3.3 Specifying a scheduling strategy

Computational tasks specify a scheduling strategy, which in turn govern their lifetimes. Computational tasks can specify their scheduling strategy along 3-dimensions (see Figure 3). The *counts* axis specifies the number of times a computational task needs to be executed. The *data driven* axis specifies that computational task needs to be scheduled for execution whenever data is available on any one of its constituent datasets. The periodicity axis specifies that computational tasks be periodically scheduled for execution at pre-defined intervals (specified in milliseconds).

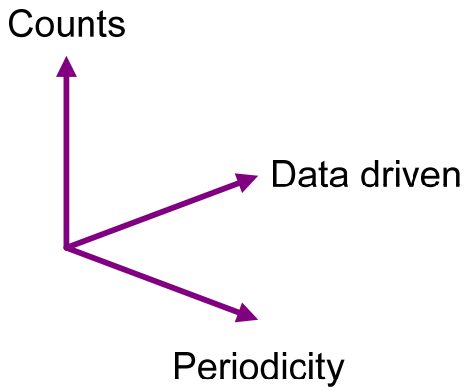

Figure 3: Dimensions for scheduling strategy

Each of these axes can extend to infinity, in which case, it constitutes a *stay-alive* primitive. A domain specialist can also specify a custom scheduling strategy that permutes along these 3-dimensions. Thus, one can specify a scheduling strategy that limits a computational task to be executed a maximum of 500 times either when data is available or at regular intervals.

A computational task can change its scheduling strategy during execution, and Granules will enforce the newly established scheduling strategy during the next *round* of execution (section 3.5). This scheduling change can be a significant one – from data driven to periodic. The scheduling change could also be a minor one with changes to the number of times the computation needs to be executed or an update to the periodicity interval.

In addition to the aforementioned primitives, another primitive – *stay alive until termination condition reached* – can be specified. In this case, the computational task continues to be stay alive until the computational task asserts that its termination condition has been reached. The termination condition overrides any other primitives that may have been specified and results in the garbage collection of the computational task.

3.4 Finite state machine for a computational task

At a given computational resources, Granules maintains a finite state machine for every computational task. This finite state machine, depicted in Figure 4, has four states: initialize, activated, dormant, and terminate.

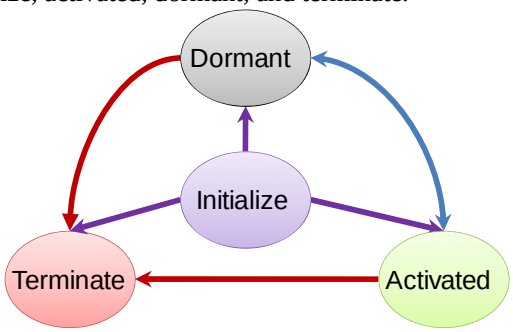

Figure 4: FSM for a computational task

The transition triggers for this finite state machine include external requests, elapsed time intervals, data availability, reset counters and assertions of the termination condition being reached.

When a computational task is first received in a deployment request, Granules proceeds to initialize the computational task. The finite state machine created for this computational task starts-off in the *initialize* state.

If, for some reason, the computational task cannot proceed in its execution, either because the datasets are not available or the start-up time has not yet elapsed, the computational task transitions into the *dormant* state. If there were problems in initialization, the computational task transitions into the *terminate* state.

If, on the other hand, the computational task was initialized successfully, and is ready for execution with accessible datasets, the computational task transitions into the *activated* state.

3.5 Interleaving execution of computational tasks

At each computational resource, Granules maintains a pool of worker threads to manage and interleave the concurrent execution of multiple computational tasks.

When a computational task is activated and ready for execution, it is moved into the activated queue. As, and when, worker threads become available the computational tasks are pulled from the FIFO queue and executed in a separate thread. Upon completion of the computational task, the worker thread is returned back to the thread-pool, to be used to execute other pending computational tasks within the activated queue. The computational task is placed either in the dormant queue or scheduled for garbage collection depending on the state of its finite state machine.

After a computational task has finished its latest (or the first) round of execution, checks are made to see if it should be terminated. To do so, the scheduling strategy associated with the computational task is retrieved. If a computational task needs to execute a fixed number of times a check is made to see if the counter has reset. If the computational task specifies a stay-alive primitive based either on data-availability or periodicity, checks are made to see if the datasets continue to be available or if the periodicity interval has elapsed. A check is also made to see if the computational task has asserted that its termination condition has been reached.

If none of these checks indicate that the computational task should be terminated, the computational task is scheduled for another round of execution or the computational task transitions into the dormant state. A computational task can continually toggle between the dormant and activated state till a termination condition has been reached.

3.5.1 Sizing thread-pools

The number of worker threads within the thread-pool is configurable. In general, the number of threads needs to be balanced so that the accrued concurrency gains are not offset by context-switching overheads among the threads. As a general rule, it is a good idea to set this number to be approximately equal to the number of execution pipelines available on a given machine. Thus, for a quad-core CPU with 2 execution pipelines per core, the thread-pool will be setup to have approximately 8 threads.

3.6 Diagnostics

In Granules, a user can track the status of a specific computational task or collections (job) of computational tasks. The system maintains diagnostic information about every computational task. This includes information about the number of times a computational task was scheduled for execution, its queuing overheads, its CPU-bound time, the time it was memory-resident, and the total execution time. A computational task can also assert that diagnostic messages be sent back to the client during any (or some) of its state transitions. On the client side, an observer can be registered for collections of computational tasks to track their progress without the need to actively poll individual computational tasks.

4 Support for Map-Reduce in Granules

Map-Reduce is the dominant framework used in cloud computing settings. In Map-Reduce, a large dataset is broken-up into smaller chunks that are concurrently operated upon by map function instances. The results from these map functions (usually <*key*, *value*> pairs) are combined in the reducers which collates the values for individual keys. Typically, there are multiple reducers, and the outputs from these reducers constitute the final result. This is depicted in Figure 5.

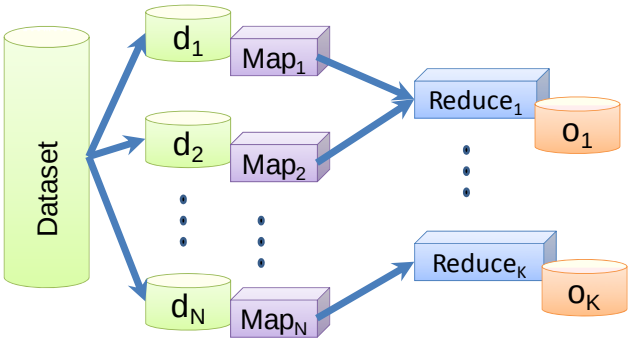

Figure 5: The basic Map-Reduce framework

The Map-Reduce framework has several advantages. First, the domain scientist only needs to provide the Map-Reduce functionality, and the datasets. Second, it is the responsibility of the framework to transparently scale as the number of available resources, and the problem size, increases. Finally, orchestration of the concurrent data-parallel execution is managed by the framework.

In traditional Map-Reduce intermediate stages exchange results using a set of <*key*, *value*> pairs. We have incorporated support for this basic result type. But we have also incorporated support for exchange of primitive data types such as `int`, `short`, `boolean`, `char`, `long`, `float`, and `double`. We have also incorporated support for exchanging arrays (`[]`) and 2D arrays (`[][]`) of these primitive data types. There is also support for exchanging `Objects` that encapsulated compound data types, along with arrays and 2D arrays of these `Objects`.

The intermediate results in most Map-Reduce implementations utilize file IO for managing results produced by the intermediate stages. The framework then notifies appropriate reducers to *pull* or retrieve these results for further processing.

Depending on the application, the overheads introduced by performing such disk-IO can be quite high. In Granules, we use streaming to *push* these results onto appropriate reducers. Streaming, as validated by our benchmarks (described in section 6), is significantly faster and we think that there are several classes of applications that can benefit from this.

Additionally, since the results are being streamed as and when they have been computed, successive stages have access to partial results from preceding stages instead of waiting for the entire computation to complete. This is particularly useful in situations where one is interested in getting as many results as possible within a fixed amount of time.

4.1 Two sides of the same coin

In Granules, map and reduce are two roles associated with the computational task. These roles inherit all the computational task functionality, while adding functionality specific to their roles.

The map role adds functionality related to adding, removing, tracking and enumerating the reducers associated with the map function. Typically, a map function has one reducer associated with it. In Granules, we do not limit the number of reducers associated with a map function. This feature can be used to fine-tune redundancies within a computational pipeline.

The reduce role adds functionality related to adding, removing, tracking and enumerating maps associated with it. The reducer has facilities to track output generated by the constituent maps. Specifically, a reducer can determine if partial or complete outputs have been received from the maps. The reduce role also incorporates support to detect and discard any duplicate outputs that may be received.

The map and reduce roles have facilities to create and publish results. The payloads for these results can be primitive data types that we discussed earlier, `Objects` encapsulating compound data types, <*key*, *value*> pairs, arrays and 2D arrays of the same. In Granules, generated results include sequencing information and metadata specific to the generator. Additionally, an entity is allowed to assert if these results are partial results and/or if the processing has been completed.

Since map and reduce are two roles of the computational task in Granules, they inherit functionality related to scheduling strategy (and lifecycle management), diagnostic strategy and dataset management.

Individual map and reduce instances toggle between the activated and dormant states (section 3.5) till such time that it is ready to assert that its termination condition has been reached. For example, a reducer may assert that it has reached its termination condition only after it has received, and processed, the outputs of its constituent maps.

4.2 Setting up graphs

Granules supports a set of operations that allow these graphs to be setup. Individual maps can add/remove reducers. Similarly, reducers are allowed to add/remove maps. The functions are functionally equivalent. Granules also allows the map and reduce roles to be *interchangable*: a map can act as a reducer, and vice versa. Figure 6 depicts how support for addition/removal of roles combined with role inter-changeability, can be used to create a graph with a feedback loop. In our benchmarks, involving the *k-means* machine learning algorithm, we have 3 stages with a feedback loop from the output of stage 2 to its input. Granules manages overheads related to ensuring that the outputs from the map are routed to the correct reducers.

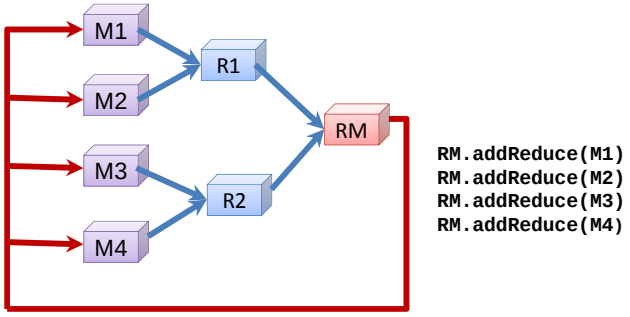

Figure 6: Creating a simple feedback loop

Additionally, Granules can create execution graphs once the number of map and reduce instances in a pipeline have been specified. Granules ensures the appropriate linkage of the map reduce instances.

4.3 Creating computational pipelines

Typically, in Map-Reduce the instances that comprise an execution pipeline are organized in a directed acyclic graph (DAG) with the execution proceeding in sequence through monotonically increasing stages.

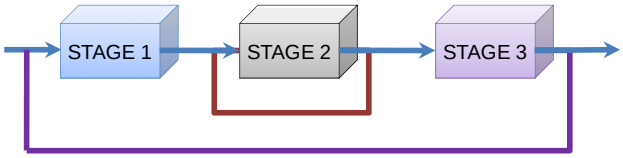

Figure 7: Creating pipelines with cycles

In Granules, we have incorporated support for cycles to be present. This allows Granules to feedback the outputs of some stage, within a pipeline, to any of its preceding stages. The system places no restrictions on the span-length, or the number, of the feedback in the pipeline. In a sense it can be argued that Granules supports both data and control flow graphs. An example of such a computational graph in Granules is depicted in Figure 7.

One feature of the computational task plays a role in allowing these loops: the notion of the stay-alive computation. Furthermore, since this is available at the micro-level (computational task), individual stages, collection of stages, or the computational pipeline itself can itself be iterative, periodic, data-driven, or termination condition dependent.

Granules manages the pipeline complexity. The domain scientist does not need to cope with *fan-in* complexity, which corresponds to the number of units that feed results into a given instance. Once a pipeline has been created, a domain specialist does not have to cope with IO, synchronization, or networking related issues. The runtime includes facilities to track outputs from preceding stages.

4.4 Observing the lifecycle of a pipeline

At the client side, during the deployment process, Granules allows a lifecycle observer to be registered for an execution pipeline. This observer processes diagnostic messages received from different computational resources running Granules. These diagnostic messages relate to state transitions associated with the different computational task instances (and the map, reduce roles) and the pertinent metrics associated with the computation task. The lifecycle observer reports to the client upon completion of an execution pipeline. The observer also reports errors in the execution of any of the units that comprise the pipeline.

5 Related Work

The original Map-Reduce paper [1] by Ghemawat and Dean described how their programming abstraction was being used in the Google search engine and other data-intensive applications. This work was itself inspired by *map* and *reduce* primitives present in Lisp and other functional programming languages. Google Map-Reduce is written in C++ with extensions for Java and Python. Swazall [7] is an interpreted, procedural programming language used by Google to develop Map-Reduce applications.

Hadoop [8] was originally developed at Yahoo, and is now an Apache project. It is by far the most widely used implementation of the Map-Reduce framework. In addition to the vast number of applications at Yahoo, it is also part of the Google/IBM initiative to support university courses in distributed computing. Hadoop is also hosted as a framework over Amazon's EC2 [9] cloud. Unlike Granules, Hadoop supports only exactly-once semantics, meaning that there is direct support within the framework for map and reduce functions to maintain state.

Hadoop uses the Hadoop Distributed File System (HDFS) files for communicating intermediate results between the map and reduce functions, while Granules uses streaming for these disseminations, thus allowing access to partial results.

HDFS allows for replicated, robust access to files. During the data staging phase, Hadoop allows creation of replicas on the local file system; computations are then spawned to exploit data locality. Hadoop supports automated recovery from failures. Currently, Granules does not incorporate support for automated recovery from failures. This will be the focus of our future work in this area: here, we plan to harness the reliable streaming capabilities available in NaradaBrokering.

The most dominant model for developing parallel applications in the HPC community is the SPMD [2] model (first proposed by Federica Derema) in tandem with the Message Passing Interface (MPI) [10] library. The SPMD model is a powerful one, and Map-Reduce can in fact be thought of as an instance of the SPMD model. The use of MPI has however, not been as widespread outside the scientific community.

Microsoft Research's Dryad [11] is a system designed as a programming model for developing scalable parallel and distributed applications. Dryad is based on directed, acyclic graphs (DAG). In this model, sequential programs are connected using one-way channels. It is intended to be a super-set of the core Map-Reduce framework. Dryad provides job management and autonomic capabilities, and makes use of the Microsoft Shared Directory Service. However, since Dryad is developed based on DAGs it is not possible to develop systems that have cycles in them. For example, in our benchmarks, we were not able to implement the *k-means* machine learning algorithm [12] using the basic Dryad framework.

Phoenix [13] is an implementation of MapReduce for multi-core and multiprocessor systems. A related effort is Qt Concurrent [14], which provides a simplified implementation of the Map-Reduce framework in C++. Qt Concurrent, automatically optimizes thread utilizations on multi-core machines depending on core availability. Disco [15], from Nokia, is an open source Map-Reduce runtime developed using the Erlang functional programming language. Similar to the Hadoop architecture, Disco stores the intermediate results in local files and accesses them using HTTP connections from the appropriate reduce tasks.

Holumbus [16] includes an implementation of the Map-Reduce framework, developed in the Haskell functional programming language at the FH Wedel University of Applied Sciences, Germany.

Skynet [17] is an open-source Ruby based implementation of the Map-Reduce framework. Skynet utilizes a peer-recovery system for tracking the constituent tasks. Peers track each other and, once failure is detected, can spawn a replica of the failed peer.

We had originally developed a prototype implementation of Map-Reduce, CGL-MapReduce [18], which implemented Map-Reduce using streaming (once again, using NaradaBrokering) with the ability to keep-alive map instances. Granules represents an overhaul, and incorporates several new capabilities such as built-in support for sophisticated lifecycle management (periodicity, data driven and termination conditions), powerful creation and duplicate detection of results, and diagnostics in addition to the ability to create complex computational pipelines with feedback loops in multiple stages. The code-base for the Granules (available for download) runtime has also been developed from scratch.

6 Benchmarks:

In our benchmarks we profile several aspects of the Granules' performance. We are specifically interested in determining system performance for different lifecycles associated with the computational tasks. The different lifecycles we benchmark include exactly-once, iterative, periodic and data driven primitives. Where possible, we contrast the performance of Granules with comparable systems such as Hadoop, Dryad and MPI. It is expected

that these benchmarks would be indicative of the performance that can be expected in different deployments.

All machines involved in these benchmarks have 4 dual-core CPUs, 2.4GHz clock and 8GB of RAM. These machines were hosted on a 100 Mbps LAN. The Operating System on these machines is Red Hat Enterprise Linux version 4. All Java processes executed within version 1.6 of Sun's JVM. We used version 3.4.6 of the gcc complier for C++, and for MPI we used version 7.1.4 of the Local Area Multicomputer (LAM) MPI [19].

6.1 Streaming Substrate

Since we use the NaradaBrokering streaming substrate for all communications between entities, we present a simple benchmark to give the reader an idea of the costs involved in streaming. Our results outline the communication latencies in a simplified setting involving one producer, one consumer and one broker. The communication latencies are reported for stream fragments with different payload sizes. Additional NaradaBrokering benchmarks in distributed settings can be found in [4,5].

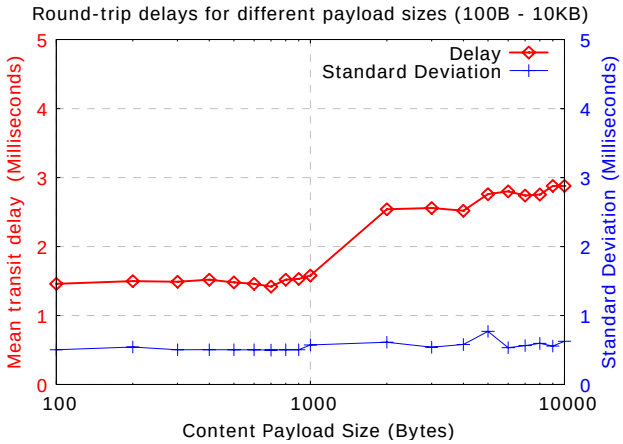

Figure 8: Streaming overheads in cluster settings

Two cluster machines were involved in this benchmark. The producer and consumer were hosted on the same machine to obviate the need to account for clock drifts while measuring latencies for streams issued by the producer, and routed by the broker (hosted on the second machine) to the consumer.

The reported delay, in the results depicted in Figure 8, is the average of 50 samples for a given payload size; the standard deviation for these samples also being reported. The Y-axis for the standard deviation is the axis on the right-side (blue) of the graph. Streaming latencies vary from 750 microseconds per-hop for 100 bytes to 1.5 milliseconds per-hop for a stream fragments of 10 KB in cluster settings.

6.2 Information Retrieval: Exactly once

In this sub-section we present results from a simple information retrieval example. Given a set of text files, the objective is to histogram the counts associated with various words in these files. The performance of Granules is contrasted with that of Hadoop and Dryad. The Dryad version that we have access to uses C#, LINQ and file-based communications using the Microsoft Shared Directory service. The OS involved in the Dryad benchmarks is Windows XP.

For this benchmark, we vary the cumulative size of the datasets that need to be processed. The total amount of data that is processed is varied from 20 GB to 100 GB. There were a total of 128 map instances that were deployed on the 5 machines involved in the benchmark.

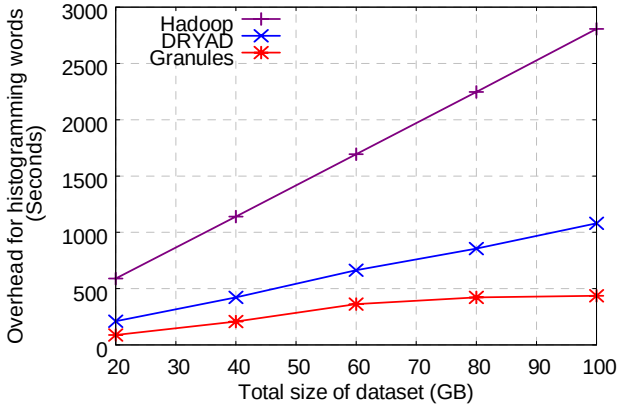

Figure 9: Processing time for histogramming words

The results, depicted in Figure 9, demonstrate the benefits of using streaming as opposed to file-based communications. As the size of datasets increases, there is a concomitant increase in the number and size of the intermediate results (file-based). This contributes to the slower performance of Hadoop and Dryad. We expect the performance of Dryad's socket-based version to be faster than their file-based version.

6.3 K-means: Iterative

Machine learning provides a fertile ground for iterative algorithms. In our benchnmarks, we considered a simple algorithm in the area of unsupervised machine learning: *k-means*. Given a set of *n* data points, the objective is to organize these points into *k* clusters.

The algorithms starts-off by selecting *k* centroids and then associates different data points within the dataset to one of the clusters based on their proximity to the centroids. For each of the clusters, new centroids are then computed. The algorithm is said to converge when the cumulative Euclidean distance between the centroids in successive iterations is less than a pre-defined threshold.

In *k-means*, the number of iterations depends on the initial choice of the centroids, the number of data points and the specified error rate (signifying that the centroid movements are acceptable). The initial set of data points is loaded at each of the map functions. Each map is responsible for processing a portion of the entire dataset. What changes from iteration to iteration are the centroids. The output of each map function is a set of centroids.

The benchmarks, which were run on 5 machines, also contrast the performance of Granules with MPI using a C++ implementation of the *k-means* algorithm.

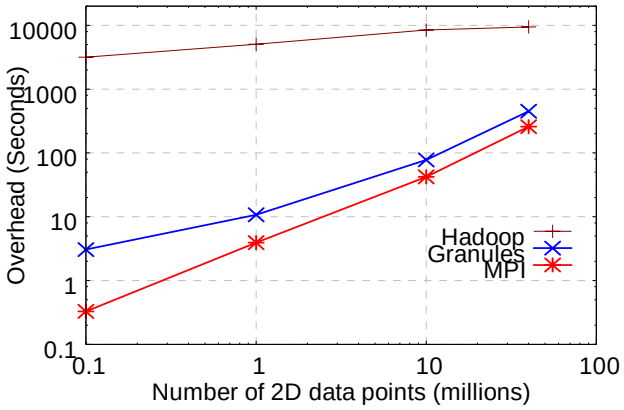

Figure 10: Performance of the k-means algorithm

The graphs, depicted in Figure 10, have been plotted on a log-log graph so that the trends can be visualized a little better. We varied the number of data points in the dataset from 10^5 to $4x10^7$. The results indicate that Hadoop's performance is orders of magnitude slower than Granules and MPI. In Hadoop, these centroids are transferred using files, while Granules uses streaming. Furthermore, since Hadoop does not support iterative semantics, map functions need to be initialized and the datasets need to be reloaded using HDFS. Though these file system reads are being performed locally (thanks to HDFS and data collocation) these costs can still be prohibitive as evidenced in our benchmarks. Additionally, as the size of the dataset increases, the performance of the MPI/C++ implementation of *k-means* and the Granules/Java implementation of *k-means* start to converge.

6.4 Periodic scheduling

In this subsection we benchmark the ability of Granules to periodically schedule tasks for execution. For this particular benchmark, we initialized 10,000 map functions that needed to be scheduled for execution every 4 seconds.

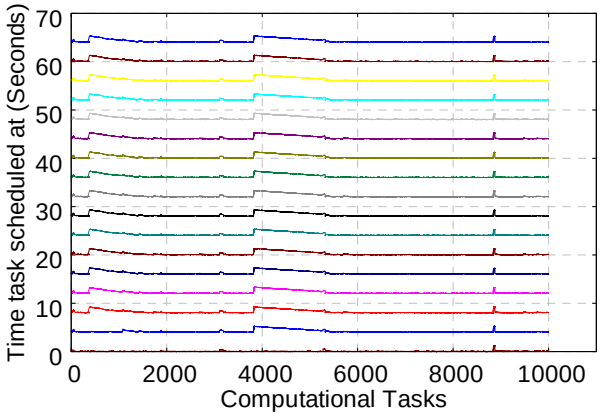

Figure 11: Periodic scheduling of 10,000 computational tasks

The objective of this benchmark is to show that a single Granules instance can indeed enforce periodicity for a reasonable number of map instances.

Figure 11 depicts the results of periodic executions of 10,000 maps for 17 iterations. The graph depicts the spacing in the times at which these maps are scheduled for execution. The X-axis represents a specific map instance (assigned IDs from 1 to 10,000) and the Y-axis represents the spacing between the times at which a given instance was scheduled. Each map instance reports 17 values.

The first time a computational task is scheduled for execution, a base time t_b is recorded. Subsequent iterations report the difference between the base time t_b and the current time, t_c. In almost all cases, the spacing between the successive executions for any given instance was between 3.9-4.1 seconds. In some cases, there is a small notch; this reflects cases where the first execution was delayed by a small amount, the (constant) impact of which is reflected in subsequent iterations for that map instance.

6.5 Data driven

In this subsection, we describe the performance of matrix multiplication using Granules. In this case, the object is to measure the product of two dense 16000 x 16000 matrices i.e. each matrix has 256 million elements with predominantly non-zero values.

The matrix multiplication example demonstrates how computational tasks can stay alive, and be scheduled for execution when data is available. The maps are scheduled for execution as and when data is available for the computations to proceed.

For this benchmark, we vary the number of machines involved in the experiment from 1 to 8. There are a total of 16000 map instances. At a given time, each of these maps process portions of the rows and columns that comprise the matrix. Each Granules instance copes with fragment of more than 2000 concurrent streams. In total, every Granules instance copes with 32000 distinct streams.

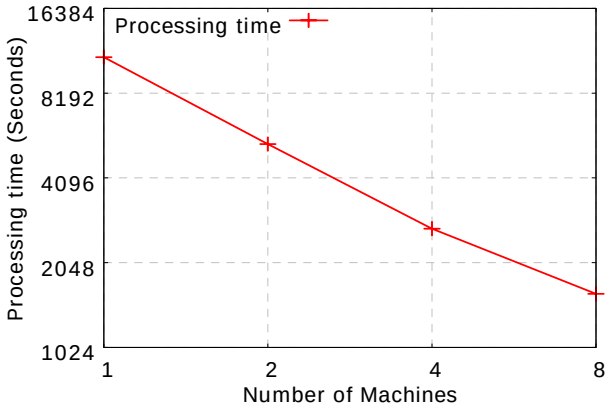

Figure 12: Processing time for matrix multiplication on different machines

The results for the processing times (plotted on a log-log scale) can be seen in Figure 12. In general as the number of available machines increase, there is a

proportional improvement in the processing time. Our plots of the speed-up (Figure 13) in processing times with the availability of additional machines reflect this.

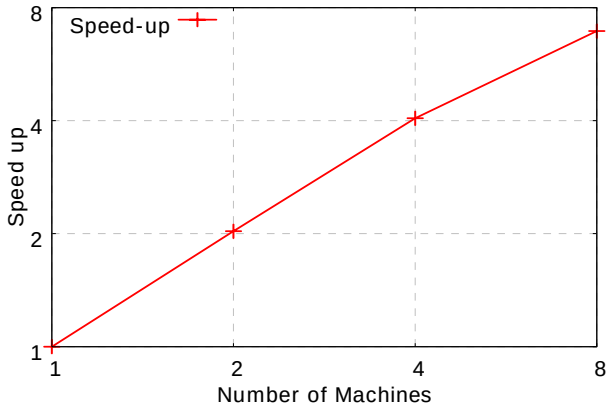

Figure 13: Speed-up for matrix multiplication

In general, these graphs demonstrate that Granules can bring substantial benefits to data driven applications by amortizing the computational load on a set of machines. Domain scientists do not need to write a single-line of networking code, Granules manages this in a transparent fashion for the applications.

6.6 Assembling mRNA sequences

This sub section describes the performance of Granules in orchestrating the execution of applications developed in languages other than Java. The application we consider is the CAP3 [20] messenger Ribonucleic acid (mRNA) sequence assembly application (C++) developed at Michigan Tech.

As Expressed Sequence Tag (EST) corresponds to mRNAs transcribed from genes residing on chromosomes. Individual EST sequences represent a fragment of mRNA. CAP3 allows us to perform EST assembly to reconstruct full-length mRNA sequences for each expressed gene.

Our objective as part of this benchmark was also to see how Granules can be used to maximize core utilizations on a machine. CAP3 takes as input a set of files. In our benchmark we need to process 256 files during the assembly.

On a given machine, we fine tuned the concurrency by setting the number of worker threads within the thread pool to different values. By restricting the number of threads, we also restricted the amount of concurrency and the underlying core utilizations. We started off by setting the worker pool size to 1, 2, 4 and 8 on 1 machine, and then used 8 worker threads on 2, 4 and 8 machines. This allowed us to report results for 1, 2, 4, 8, 16, 32 and 64 cores.

The results of our benchmark in terms of processing costs and the speed-ups achieved are depicted in Figure 14 and Figure 15 respectively. In general as the number of available cores increase, there is a corresponding improvement in execution times.

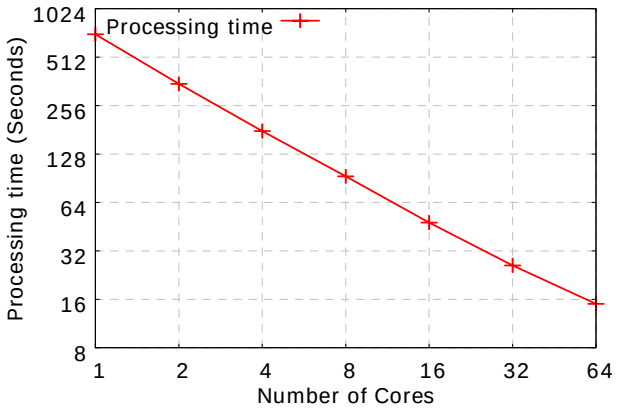

Figure 14: Processing time for EST assembly on different cores using Granules and CAP3

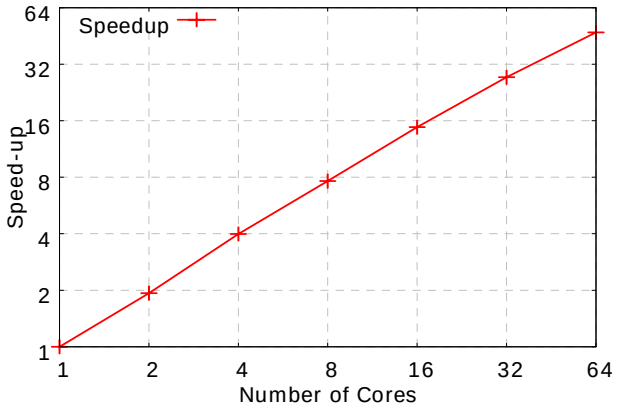

Figure 15: Speed-up for EST assembly using Granules and CAP3

The results demonstrate that, when configured correctly, Granules can maximize core utilizations on a given machine. The graphs, plotted on a log-log scale indicate that for every doubling of the available cores, the processing time for assembling the mRNA sequences reduces by half (approximately). Currently, the Granules runtime ready the thread-pool sizing information from a configuration file; we will be investigating mechanisms that will allow us to dynamically size these thread-pools.

7 Conclusions

In this paper, we described the Granules runtime. Support for rich lifecycle support within Granules allows computations to retain state which in turn is particularly applicable for several classes for scientific applications.

Granules allows complex computational graphs to be created. As discussed, these graphs can encapsulate both control flow and data flow. Granules enforces the semantics of complex distributed computational graphs that have one or more feedback loops. The domain scientist does not have to cope with IO, threading, synchronization or networking libraries while developing applications that span multiple stages, with multiple

distributed instances comprising each stage. These computational pipelines can be iterative, periodic, data-driven, or termination condition dependent

Demonstrable performance benefits have been accrued by Granules as result of using streaming for disseminating intermediate results.

Granules' rich lifecycle support, and its performance when contrasted with comparable systems, underscores the feasibility of using Granules in several settings. As part of our future work, we will be investigating support for autonomic error detection and recovery within Granules.

Bibliography

[1]. J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," ACM Commun., vol. 51, Jan. 2008, pp. 107-113.

[2]. F. Darema, "SPMD model: past, present and future," Recent Advances in Parallel Virtual Machine and Message Passing Interface, 8th European PVM/MPI Users' Group Meeting, Santorini/Thera, Greece, 2001.

[3]. S. Pallickara, J. Ekanayake, and G. Fox. "An Overview of the Granules Runtime for Cloud Computing". (Short Paper) Proceedings of the IEEE International Conference on e-Science. Indianapolis. 2008.

[4]. S Pallickara and G Fox. "NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids." Proceedings of the ACM/IFIP/USENIX International Middleware Conference Middleware-2003. pp 41-61.

[5]. S Pallickara et al. "A Framework for Secure End-to-End Delivery of Messages in Publish/Subscribe Systems". Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID 2006). Barcelona, Spain.

[6]. S. Pallickara, H. Bulut, and G. Fox, "Fault-Tolerant Reliable Delivery of Messages in Distributed Publish/Subscribe Systems," 4th IEEE International Conference on Autonomic Computing, Jun. 2007, pp. 19.

[7]. R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with Sawzall," .Scientific Programming Journal Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure, vol. 13, no. 4, pp. 227–298, 2005.

[8]. Apache Hadoop, http://hadoop.apache.org/core/

[9]. S. Garfinkel An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS . Tech. Rep. TR-08-07, Harvard University, August 2007.

[10]. Message Passing Interface Forum. MPI: A Message Passing Interface. In Proc. of Supercomputing '93, pages 878–883. IEEE Computer Society Press, November 1993.

[11]. M. Isard , M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," European Conference on Computer Systems , March 2007.

[12]. J. B. MacQueen (1967): "Some Methods for classification and Analysis of Multivariate Observations, Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability", Berkeley, University of California Press, 1:281-297

[13]. C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. "Evaluating MapReduce for Multi-core and Multiprocessor Systems," Proc. International Symposium on High-Performance Computer Architecture (HPCA), 2007, pp. 13-24.

[14]. Qt Concurrent. Simplified MapReduce in C++ with support for multicores. http://labs.trolltech.com/page/Projects/Threads/ QtConcurrent. April 2009.

[15]. Disco project, http://discoproject.org/

[16]. S. Schlatt, T. Hübel, S. Schmidt and U. Schmidt. The Holumbus distributed computing framework & MapReduce in Haskell. http://holumbus.fh-wedel.de/trac. 2009.

[17]. A Pisoni. Skynet: A Ruby MapReduce Framework. http://skynet.rubyforge.org/. April 2009.

[18]. J. Ekanayake, S Pallickara and G. Fox. "Map-Reduce for Scientific Applications". Proceedings of the IEEE International Conference on e-Science. Indianapolis. 2008.

[19]. J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In Proceedings, Euro PVM/MPI, October 2003.

[20]. X. Huang and A. Madan (1999). CAP3: A DNA Sequence Assembly Program. Genome Research, 9: 868-877.