Brasil

Basic Resource Aggregation System Infrastructure Layer

Eric Van Hensbergen
IBM Research
bergevan@us.ibm.com

Pravin Shinde
ETH Zurich
shindep@student.ethz.ch

Noah Evans
Alcatel-Lucent Bell Labs
npe@plan9.bell-labs.com

ABSTRACT

Brasil is a self-contained service which can be deployed across a cluster to provide a dataflow workload distribution and communication aggregation mechanism. Together with our dataflow shell, named PUSH, it is intended to be used for the management of non-traditional super computing applications as well as provide a mechanism to manage in-situ analysis and vizualization of more traditional high performance computing simulations. This paper describes our experiences implementing and deploying a prototype of Brasil on a BlueGene/P supercomputer.

1. INTRODUCTION

The deluge of huge data sets such as those provided by sensor networks, online transactions, and petascale simulation provide exciting opportunities for data analytics. The scale of the data makes it increasingly difficult to process in a reasonable amount of time on isolated machines. In the near future petascale and exascale simulation will make the involvement of secondary storage and disks impractical, driving the need for infrastructures which provide in-situ analytics and visualization capabilities [8].

This has lead to data flow systems emerging as the standard tool for solving research problems using these vast datasets. In typical dataflow systems, runtimes [3] [1] [7] define graphs of processes, the edges of the graphs representing pipes and their vertices representing computation on a system. Within these runtimes a new class of languages [10] [12] [9] can be used by researchers to solve "pleasantly parallel" problems (problems where the individual elements of datasets are considered to be independent of any other element) more quickly without worrying about explicit concurrency.

These languages provide automated control flow(typically matched to the architecture of the underlying runtime) and channels of communication between systems. In existing systems, these workflows and the underlying computation are tightly linked, tying solutions to a particular runtime, workflow and language. This creates difficulties for analytics researchers who wish to draw upon tools written in many different languages or runtimes which may be available on several different architectures or operating systems.

Our experiences with existing tools for constructing workflows for simulation, transformation, analysis and visualization were frustrating. This was primarily due to the tight coupling of language, runtime, and workflow tools which proved difficult to integrate with existing applications. We also found many of these systems difficult (if not impossible) to deploy on petascale clusters, particularly those with dynamic resources such as clouds where nodes might be added or removed based on load, failure, or different system priorities.

We observed that UNIX pipes were perfectly designed to allow developers to hook together tools written in different languages and runtimes in ad-hoc fashions. This allowed tool developers to focus on doing one thing well, and enabled code portability and reuse in ways not originally conceived by the tool authors. The UNIX shell incorporated a model for tersely composing these smaller tools into pipelines (e.g. 'sort | uniq -c'), creating a coherent workflow to solve more complicated problems quickly. Tools read from standard input and wrote to standard output, allowing programs to work together in streams *with no explicit knowledge of this chaining built into the program itself.*

One to one pipelines such as those used by a typical UNIX shell, can not be trivially mapped to streaming workflows which incorporate one-to-many, many-to-many, and many-to-one data flows. UNIX shell pipelines are strictly local, so while they can be used in conjunction with distributed system tools such as ssh, they do not themselves naturally facilitiate networked operation. Additionally, typical UNIX pipeline tools write data according to buffer boundaries instead of record boundaries.

To address these issues we have implemented a prototype shell, which we call PUSH, using dataflow principles and incorporating extended pipeline operators to establish distributed workflows —potentially running on clusters of machines— and correlate results. In order to scale this approach to large clusters of servers we implemented a workload and resource distribution infrastructure which incorperated dataflow communication constructs which we call *Brasil.*

The rest of this paper is focused on describing the design and implementation of Brasil. The next section covers some additional details of the PUSH shell and describes some of the key design elements of Brasil motivated by it. Section

3 discusses our prototype implementation in more detail including lessons learned while attempting to scale the prototype to thousands of cores. Section 4 contains our evaluation of the overhead of the infrastructure when deployed on leadership class high-performance computing environments. In Section 5 we will conclude by discussing potential improvements to address the overheads and explore opportunities for improvement in both the design and implementation of our approach.

2. DESGIN

The PUSH shell is a conventional UNIX shell with two additional pipeline operators, a multiplexing fan-out(`|<`[n]), and a coalescing fan-in(`>|`). This combination allows PUSH to distribute I/O to and from multiple simultaneous threads of control. A fan-out argument, n, specifies the desired degree of parallel threading. If no argument is specified, the default of spawning a new thread per record (up to the limit of available cores) is used. This can also be overriden by command line options or environment variables. The pipeline operators provide implicit grouping semantics allowing natural nesting and composibility. While their complimentary nature usually lead to symmetric mappings (where the number of fan-outs equal the number of fan-ins), there is nothing within our implementation which enforces it. Normal redirections as well as application specific sources and sinks can provide alternate data paths.

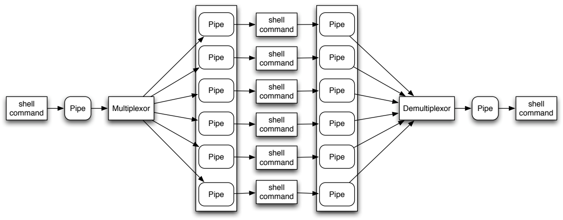

Figure 1: The structure of the PUSH shell

PUSH also differs from traditional shells by implementing native support for record based input handling over pipelines. This facility is similar to the argument field separators, IFS and OFS, in traditional shells which use a pattern to determine how to tokenize arguments. PUSH provides two variables, ORS and IRS, which point to record separator modules. These modules (called multiplexors in PUSH) split data on record boundaries, emitting individual records that the system distributes and coalesces. The choice of which *multipipe*, an ordered set of pipes, to target is left as a decision to the module.

Different data formats may have different output requirements. Demultiplexing from a multipipe is performed by creating a many to one communications channel within the shell. The shell creates a reader processes which connects to each pipe in the multipipe. When the data reaches an appropriate record boundary a buffer is passed from the reader to the shell which then writes each record buffer to the output pipeline.

An example from our particular experience, Natural Language Processing, is to apply an analyzer to a large set of files, a "corpus". User programs go through each file which contain a list of sentences, one sentence per line. They then tokenize the sentence into words, finding the part of speech and morphology of the words that make up the sentence. There are a large number of discrete sets of data whose order is not necessarily important. We need to perform a computationally intensive task on each of the sentences, which are small, discrete records an ideal target for parallelization.

PUSH was designed to exploit this mapping. For example, to get a histogram of the distribution of Japanese words from a set of documents using chasen, a Japanese morphological analyzer, we take a set of files containing sentences and then distribute them to a cluster of machines on our network. The command is as follows:

```
ORS=blm find . -type f |< \\
xargs chasen | sort | uniq -c >| sort -rn
```

The first variable, ORS, declares our record multiplexor module, the intermediary used to ensure that the input and output to distributed pipes are correctly aligned to record boundaries. In many cases a default which splits based on atomic writes can be used. Alternatively one of several built-ins which split based on a field separator or newline can be used.

find . -type f gives a list of the files which are then "fanned out"(`|<`) using a combination of a multipipes, and a *multiplexor* which determines which pipes are the targets of each unit of output. This fanned out data goes to xargs on new threads which then uses the filenames as arguments to chasen. The find acts as a command driver, fanning out file names to the individual worker machines. The workers then use the filenames input to xargs, which uses the input filenames as arguments to the target command. Using the output of the analyzer (Japanese words) are then sorted and counted using uniq. Finally these word counts are "fanned in"(`>|`) to the originating machine which then sorts them.

While PUSH works perfectly well on a stand-alone machine, the original intent was to use it to drive workloads and workflows across a cluster of machines. In particular, we were interested in deploying on leadership class high performance computing machines and as such a key requirement for us was scalability to a large number of nodes. Instead of traditional client/server models we opted for a peer based model where every node within the system is capable of initiating new workflows of computation and managing the newly created pipeline. This allows workflows and component applications to initiate new branches of computation or analysis at any stage of the pipeline or in reaction to data produced by a previous portion of the pipeline giving the entire infrastructure a degree of elasticity that seemed to be missing from previously available tools. In order to accomplish this we designed the system without any central component which required knowledge of the entire system. All knowledge is distributed and then aggregated at certain points. Descisions like scheduling and job management are also made in a distributed fashion, utilizing the hierarchy of aggregation points to eliminate the need for all-to-all communication during workload distribution.

In order to scale such a system we decided to go with a hierarchical organization of nodes. Aggregation points scale command and communication between nodes. When a node or its children have insufficient resources to satisfy a request, the request is propagated to the parent aggregation point. This form of hierarchical aggregation also seems to map well with the physical topology of many leadership class systems.

The hierarchical topology not only helps scale communication and control, but it also can help users and application components traverse multiple network domains in a seamless fashion. We intend to use this property to create an environment which seamlessly extends the user's desktop experience and environment to the supercomputer.

The forms of communication aggregation match the new pipeline primatives which are part of the PUSH shell. By default, input to each master pipeline component thread is fanned out to sub-session threads and output from those sub-sessions in fanned back into the master component. It is intended that this behavior be setup and controlled by the Brasil infrastructure which spawns the sub-session threads (whether they be local or remote). For multi-stage pipelines, we need the ability to direct the output of a pipeline component to another pipeline component without necessarily passing that data through the master thread (for efficiency reasons). In order to accomplish that we've added the ability to splice input or output of individual subsessions threads to eachother.

3. IMPLEMENTATION

This section describes the internals of the implementation in more detail. It is important to remember that these details are entirely hidden by the PUSH shell so users and their applications are never really exposed to the complexity expressed herein. It is possible to provide alternate front-ends to the underlying Brasil infrastructure and applications may choose to interact directly with it and its control interfaces, but it is not expected that this will be the default mode of operation.

3.1 Interface

The external interface to our infrastructure is a synthetic file system much like the interface to the Xcpu [6] workload distribution system. Figure 2 gives the the high-level view of the hierarchy of the Brasil namespace.

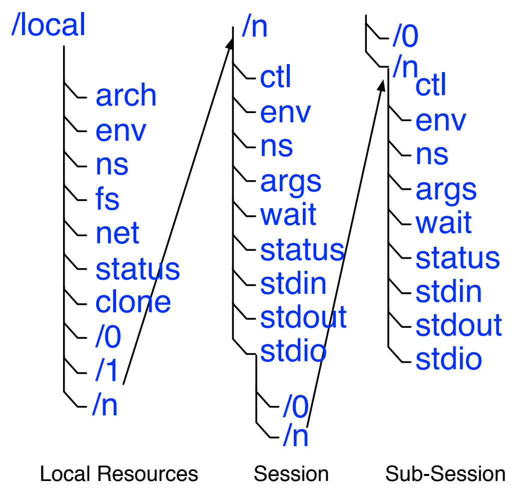

Figure 2: Filesystem interface in Brasil

Each node in the system has a *local* directory in the root of their namespace which contains pointers to the resources available on the node and control files which allows managing jobs deployed on the node. A few files here provide information about the system, like the `[arch]` file which tells the architecture and operating system of the node while the `[status]` file provides information about total amount of resources currently available, including the remote resources which are directly or indirectly decedents of this node. Files like `[env]` and `[ns]` allow controlling the default environment and the namespace of the processes created on this node. `[fs]` and `[net]` are links to the local filesystem and networking resources available on the node. The `[clone]` file provides the interface to create new sessions which is the unit of the workload management.

The sessions are represented as directories with the session-id number as the directory name. Each directory is self-contained and provides interfaces to manage the execution of the session. Files like `[env]` and `[ns]` are present in the session directory also, and can be used to overwrite the default environment and namespace specifically for this session. The `[ctl]` file is used for controlling the execution and the `[stdio]` file is used to manipulate standard input and output.

Since sessions themselves can have children, each session directory may have subsession directories with similar files. The files at the session directory level provide the aggregate interfaces to the sub-session directory.

3.2 Brasil

The core of Brasil is a self-contained daemon which is based on a fork of Inferno [4]. Inferno is an open source distributed operating system which is a direct descendant of the Plan 9 operating system [11]. It runs natively on multiple hardware platforms and can also run as a user-space application on top of other operating systems. Brasil is based on the hosted version of the Inferno platform.

The Brasil namespace is exported using the `9P` protocol. This protocol is used by Plan 9 and Inferno extensively to access any file. Recently the Linux kernel has added support for the `9P` protocol[5]. This allows Linux to mount filesystems exported over the 9P protocol. Other Unix based operating systems can use FUSE for accessing the 9P based filesystems.

The applications interact with this mounted Brasil filesystem using the native filesystem interfaces (e.g. VFS for Linux). Any interaction with this Brasil filesystem is communicated to Brasil using *9P*. If needed, it uses services from the host operating system or from other Brasil filesystems deployed on remote locations. The Brasil filesystem then sends the prepared response over 9P. The host operating system will relay this response back to the application via the local filesystem interface.

3.3 Central Services

The ability to configure many Brasil nodes into hierarchy is provided by the *central services*. Contrary to the name, central services is highly distributed and every Brasil runs an independent instance of the central services.

The central services synthetic file server which provides a simple hierarchy of directory mount points representing remote nodes. Mounts of the remote nodes or binds of previously mounted remote nodes are accomplished within this file system such that anyone who mounts our namespace can also see (and access) anyone we have mounted transitively, in such a way a child node can access a parent nodes, other children, or the parents nodes parent and so forth. Any node could establish themselves within a hierarchy by binding a parent's central service directory to the name [/csrv/parent] and then tell the parent to back-mount their name space (allowing two way traversal). In this way children register with parents triggering the cross-mounts and establishing a two way link between them. Each Brasil instance needs only to know the information about its parent and children in the hierarchy and all Brasil nodes initiate these connections leading to the distributed creation of the Brasil node hierarchy.

Figure 3 tries to give a simple overview of how this synthetic filesystem view is populated based on the underlying mount connections between the nodes.

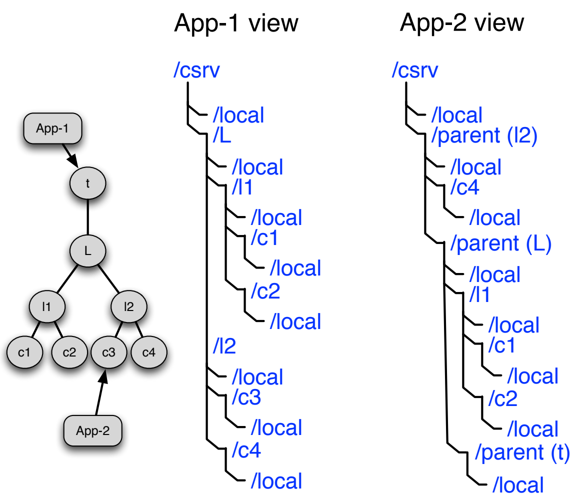

Figure 3: Sample filesystem interface for sample the topology in Brasil

Assuming the links between the nodes are created by the remote node mounts in central services, this diagram shows how the filesystem views at different nodes encompasses the whole network, even though each node is only connected to its neighbours. The Brasil filesystem starts with the [/csrv] directory. The location [/csrv/local/] presents the local resources whereas [/csrv/parent/] presents the Brasil filesystem of the parent node. All other directories in the represents the Brasil filesystem of the children nodes. It can be easily seen that both *App-1* and *App-2* have access to all the nodes even though they are running on different nodes. In this design, every node has to worry about only its children and the parent, other topology falls into place automatically.

Just because every node can construct the global view, does not mean that it must use this global view for making any decision or performing a typical operation. The nodes mostly use only the local view for decision making and operations. This local view includes the parent node and the children nodes.

3.4 Examples

While the PUSH shell handles much of the complexity of interacting with the Brasil infrastructure, it is useful to see examples of how it interacts with the underlying infrastructure in order to understand the various mechanisms better. These examples are given from the perspective of directly interacting with the infrastructure file systems from a normal UNIX shell.

The first example presents how the default aggregation behaviour of Brasil can used to deploy large number of applications.

```
$ less ./mpoint/csrv/local/clone
0
```

The above command is an example of creating a new session. The contents read from the [clone] file represent the session-ID. Now we use session 0 for performing actual execution.

```
$ echo "res 4" > ./mpoint/csrv/local/0/ctl
$ echo "exec date" > ./mpoint/csrv/local/0/ctl
$ cat ./mpoint/csrv/local/0/stdio
Fri May 7 13:53:58 CDT 2010
Fri May 7 13:53:58 CDT 2010
Fri May 7 13:53:58 CDT 2010
Fri May 7 13:53:58 CDT 2010
$
```

The first echo command sends the request for reserving 4 remote resources. The next echo command submits the request for executing the `date` command. And the `cat` command on [stdio] returns the aggregated output to the user. This example shows all the complexities about finding, connecting and using the remote resources is hidden behind the filesystem interface. This approach can be used in the *trivially parallelizable applications* where the same application is deployed on all the nodes.

When constructing more complicated dataflow pipelines, Brasil handles reserving the resources and setting up the pipe endpoints of the pipeline components. Instead of interacting with the aggregation points, dataflow applications (such as the PUSH shell) can interact directly with the sub-sessions responsible for each pipeline component.

In this example, we will try to create a small pipeline of two commands `date | wc`. But we will create this pipeline across multiple nodes.

Lets assume that session 0 is created by opening [clone] file as shown in the previous example. The following commands will create the desired pipeline.

```
$ echo "res 2" > ./mpoint/csrv/local/0/ctl
$ echo "exec date" > ./mpoint/csrv/local/0/0/ctl
$ echo "exec wc" > ./mpoint/csrv/local/0/1/ctl
$ echo "xsplice 0 1" > ./mpoint/csrv/local/0/ctl
$ cat ./mpoint/csrv/local/0/1/stdio
1 6 29
$
```

The first command [exec date] is sent to 0'th sub-session and the second command [exec wc] is sent to the 1st sub-session. The [xsplice 0 1] request tells the parent session to redirect the output of the 0'th session to the input of the 1st session. The **xsplice** command can be seen as a pipe operator of the shell script for redirecting the output of one command to the input of other command.

The above example is equivalent of executing `date | wc` on the shell, but with the difference that both commands are executed on a different remote machines while sharing the same namespace.

4. OVERHEAD EVALUATION

In order to assess the overhead of our execution deployment and communication mechanisms at scale, we deployed our infrastructure on a BlueGene/P system with resources provided by the Department of Energy's INCITE program. For the purposes of our initial prototype we limited our evaluation to a BlueGene/P configuration with only 512 nodes (comprising 2048 cores). We run Brasil on the PowerPC Linux based controller node and on the Plan 9 based I/O and Compute nodes. The user interacts with the Brasil instance via shell scripts which interact with file system interfaces exported by Brasil and mounted under linux with the v9fs file system.

4.1 Execution

Our first objective is to meaure the overhead of deployment and execution of pipeline components as the number of components increases. Since we are only interested in the overhead of deployment we chose to execute the `date` command as our example pipeline component. This is a small application which does not require any external inputs and produces small output. While this is hardly a characteristic workload, its short execution and minimal requirements will give us a better sense of the latency overheads of the infrastructure versus measuring the performance of any particular applicaiton. Each deployment involves session creation, reservation, execution, output aggregation and termination.

Figure 4 gives an initial perspective of how Brasil performs relative to sequential performance. This graph plots the total time taken by Brasil and the hypothetical time it may take for performing the same amount of work on one machine. This graph shows that the Brasil is successfully able to exploit the parallelism for deploying the jobs quickly. The Brasil deploys 2048 jobs in 12.66 seconds whereas sequential execution would take upto 510 seconds. In the graph, the line showing sequential scaling looks exponential, but that is because number of requested executions increase exponentially.

Next, we take a closer look by instrumenting the various stages of application execution to determine the overheads of each:

1. Reservation: Create a new session, and request the reservation by writing `res n` into the session `[ctl]` file. Here `n` varies from 1 to 2048 representing the number of executions requested.
2. Execution: Request the execution by writing `exec date` into the session `[ctl]` file.
3. Aggregation: Collect the output generated by all the executions by reading the session `[stdio]` file.
4. Termination: Closing all the files and terminating the session.
5. Housekeeping: Additional time taken before, between and after above steps.

Every deployment starts with the creation of the session followed by the reservation, execution, aggregation and then ending with termination of the session. We have taken the average value over multiple runs for our analysis.

Figure 5 presents the results of deployment of the date command in the form of graph. This graph presents the breakup time for various stages of the deployment using the Brasil infrastructure.

From this graph we can observe that the session termination and the housekeeping overheads are negligible compared to the time taken by reservation, execution and aggregation. So, we can ignore these two overheads in our future evaluations. For jobs of up to 128 deployments, the reservation time dominates everything else. But for larger numbers of deployment execution and aggregation time increases rapidly while reservation time remains relatively constant. This shows that reservation time is not directly dependent on the number of deployments, whereas execution and aggregation time are directly proportional to the number of deployments.

Now, let us try to analyze why reservation time is independent of the number of deployments. The reservation process involves traversing the underlying topology tree of nodes till the reservation requirements are satisfied. All the children on the same level are traversed in parallel. This way, each level is traversed in constant time, independent of the number of nodes in that level. Another aspect of the reservation mechanism which helps here is that the amount of data written and read from the `[ctl]` file and the amount of data exchanged between nodes for communicating the reservation request is fixed in size and independent of the number of deployments requested. With these two properties, the reservation time becomes directly proportional to the depth of the tree and not with the number of nodes.

We can observe the above relation in figure 5. The reservation time remains relatively constant for deployment requests from 1 to 8. Then it sharply increases between 8 to 16 and remains almost constant for all the requests between 16 to 2048. This can be attributed to underlying cluster topology. Our experimental setup used 8 IO nodes as aggregation points in the first level. This enables satisfying the requests which are smaller than 8 executions. For larger requests, one more level needs to be traversed in the topology, introducing delays. The time taken for reservation remains almost constant between 16 and 2048 executions as all these reservation requests essentially traverses the same depth.

Now let us discuss why the same property is not exhibited by execution time or aggregation time. We have discussed in the implementation chapter that all read and write requests are performed in parallel between all the nodes in the same level. But the amount of data exchanged for aggregation and execution is not constant. This data is directly proportional to the number of nodes involved. With the increase in the number of requested deployments, the amount of data to be exchanged also increases, leading to larger aggregation time. The execution time is similarly affected as all compute nodes will try to fetch the binary of the executable from the initiating node leading to the copy of the data. These observations lead us to to conclusion that *the time taken for the execution and aggregation is directly proportional to the number of deployments requested.*

Our next evaluation involve the deployment of an executable `wc` which needs input. This command counts the number of lines, words and characters in the input file. This is an interesting case for our infrastructure as this deployment involves the distribution of inputs to all the sessions. This introduces a new stage in the deployment process in addition to the 5 stages we described in the above section. This stage will be the **input** stage and involves distributing

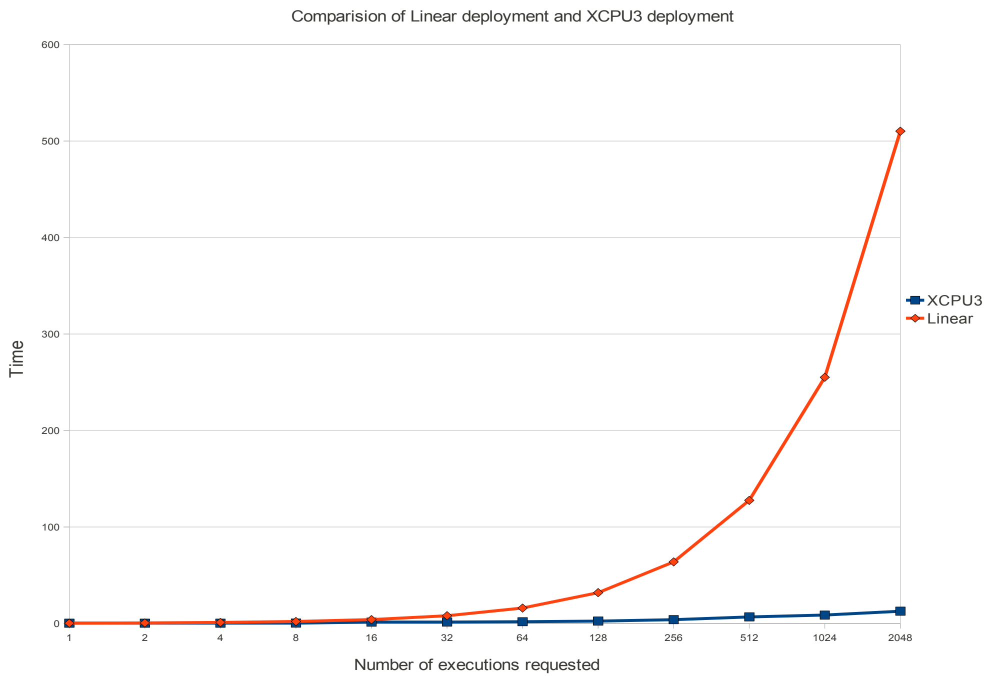

Figure 4: Comparison if Brasil with sequential deployment

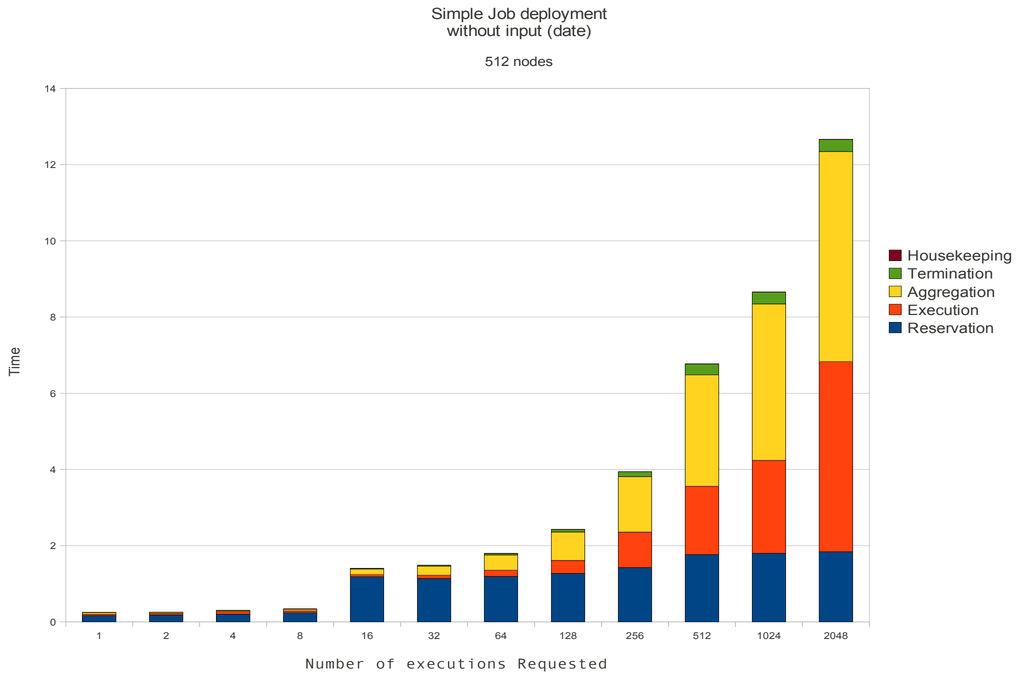

Figure 5: Deployment without input

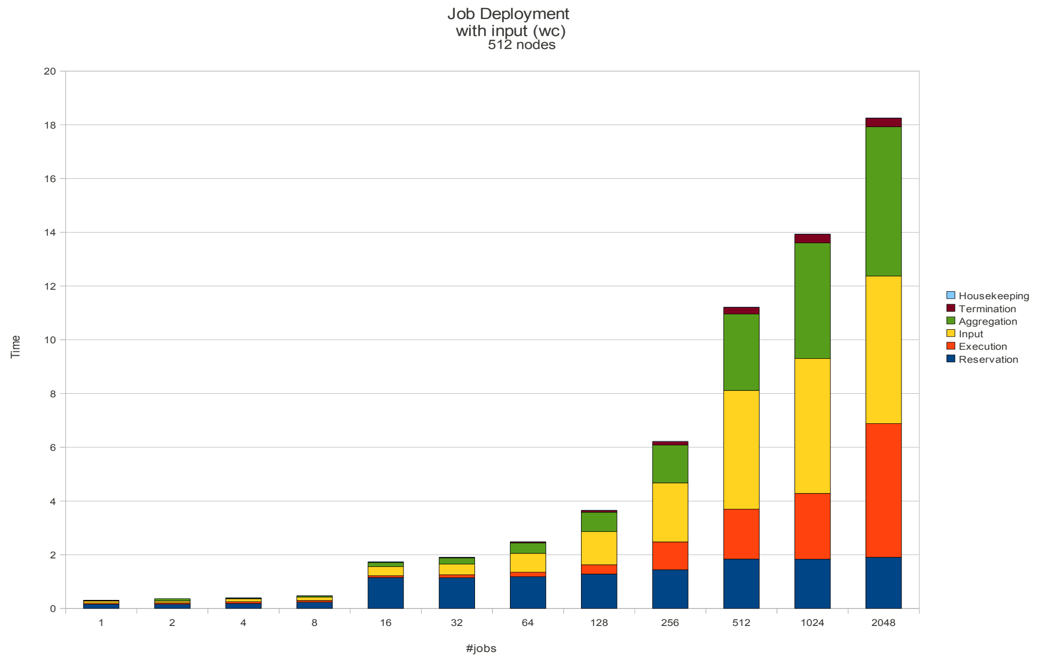

Figure 6: Deployment with input

the input data to all the sessions which are responsible for execution. As mentioned previously, Brasil will broadcast the input to all the compute nodes.

Figure 6 presents the results of evaluations involving the distribution of the input.

These results enforce our observations that reservation time is directly proportional to the the depth of the tree whereas aggregation and execution time are directly proportional to the number of deployments requested. In addition to these observations, we can also observe that the input aggregation time exhibits behavior similar to the execution and aggregation time. This observation can be attributed to the fact that input distribution implementation is similar to the output aggregation implementation.

5. DISCUSSION

While our initial experiences with using Brasil to deploy workloads was positive, we found several areas for improvement in both the design and implementation of the infrastructure.

One systemic flaw involved the design decision to give the PUSH shell responsibility for orchestrating record seperation and multiplexing. In enviornments with a lot of communication adding additional components to the I/O processing pipeline increases overhead due to increased data copying and context switching. By moving record seperation into the infrastructure itself (particularly for common default cases) we can eliminate a pipeline stage (two copies and two context switches). By moving multiplexor functionality into the infrastructure we remove an additional pipeline stage (two copies and two context switches). This reduces the overhead of interacting with the infrastructure from 6 copies and context switches down to 2. Taking advantage of techniques such as Willem de Bruijn's Beltway Buffers [2] may reduce this even further.

Another unforseen performance problem was caused by the overhead incurred by the transitive mount methodology of central services. While elegant, it introduces overhead at each transitive mount point. As such, deep hierarchies and large scale workloads can involve many transitive hops for every communication which introduces latency and load across the system. We have identified the need to augment our hierarchical aggregation (which works well for fan-out and fan-in) with cut-through models of communication for data-flow operations. There may also be opportunities to leverage the collective network capabilities of high performance systems such as BG/P to further optimize aggregate communication behavior.

While fan-in and fan-out aggregation models do match a large class of use-cases we quickly found outselves wanting more primitives to support deeper pipelines and more complex workflows. In addition to the two existing extended pipes we identified the need for deterministic delivery to/from a particular endpoint as well as many-to-many multipipes. We also quickly found the need to incorporate some form of synchronization into the infrastructure and even came up with ways of doing MPI-style collective operations using abstract pipeline constructs.

A key failure of our implementation is that it not only lacked fault tolerance, it lacked good methods of identifying where in a distributed pipeline failures actually occurred. Early debugging was plagued with workflows which would just hang waiting on input or waiting to give output. The multi-stage pipelines present within each PUSH pipeline component further complicated this. Part of these problems are inherent in the way traditional UNIX pipelines work, but we are now experimenting with "rejoinable" pipelines, task-based workqueue models, and looking into opportunities for pipeline based work-stealing and failover in order to alievate workflow stalls. Additionally, we are adding new out-of-band logging and error reporting mechanisms which attach at the same points as the I/O pipelines, but are used by the infrastructure to communicate failures.

Early on we had a lot of trouble with stalling due to trying to perform dataflow operations in a synchronous manner. We later decoupled these operations into asynchronous threads within Brasil, but this introduced a number of race conditions which made traditional pipe semantics difficult to maintain. We are left with the opinion that the only way to successfully implement multipipe semantics is to introduce control sequences to the pipelines to assist with identification of when communication begins and ends. This detracts from the elegance of pipeline based solutions, but it also prevents a host of race conditions and spurious failures. The good thing is that the complexity of dealing with these out of band control messages is hidden entirely within the infrastructure, hiding the details from the end user.

In addition to the design and implementation flaws mentioned above we realized there were several missed opportunities in the approach we took with our initial prototype. As mentioned in the evaluation section, one of the larger components of execution time is the loading of the application binary over the distributed file system. Given the hierarchical aggregation of the infrastructure we should be able to provide more efficient access to binary files and libraries. A straightforward approach is to provide cache capabilities at aggregation points, but a more aggressive concept is to incorporate the idea of collective file system pre-fetching based on session behavior utilizing underlying interconnect hardware features. Regardless of implementation, we'd like to hook distributed file system semantics and capabilities in more tightly with the infrastructure in the future.

Another set of features we overlooked in our initial implementation was adding capabilites which facilitated more dynamic behavior within sessions or within the infrastructure itself. In future implementations we plan to have a more seamless approach to nodes entering and leaving the infrastructure, which should enable cloud deployments and also ease issues with fault tolerance. In addition to dynamic behavior of the infrastructure, we also want to better support dynamic behavior within the workflow. The existing infrastructure supports spawning new workflows within the cluster as part of a workflow, but it would also be nice for workflows to be able to dynamicly request and release resources from their own workflow based on workload phase or dependencies within the data stream.

The experience of implementing the Brasil infrastructure made it clear to us that the key concept at the core of PUSH and Brasil was that of the multipipe. While we originally only used multipipes as the basis for implementing the communication aggregation primitives, it became clear to us that with a few extensions we could reimplement the control plane and monitoring subsystems in terms of multipipes as well. Given the general usefulness of the multipipe construct we feel it may be a reasonable candidate for addition to the core operating system instead of just being part of a service daemon. As such we are currently investigating implementations which add multipipes as a core system primitive to the Plan 9 and Linux operating systems and reimplementing the Brasil infrastructure using the new core system primitive.

Our current work as well as our past prototypes are avaialble as open source via http://bitbucket.org/ericvh/hare.

6. ACKNOWLEDGEMENTS

This work was supported in part by the Department of Energy Office of Science under award number DE-FG02-08ER25851.

7. REFERENCES

[1] A. Bialecki, M. Cafarella, D. Cutting, and O. O Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware, 2005. *Wiki at http://lucene. apache. org/hadoop.*

[2] Willem de Bruijn and Herbert Bos. Beltway buffers: Avoiding the os traffic jam. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, 2008.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(01):7, 2008.

[4] SM Dorward, R. Pike, DL Presotto, DM Ritchie, HW Trickey, and P. Winterbottom. The Inferno Operating System. *Bell Labs Technical Journal*, 2(1):5–18, 1997.

[5] E. Van Hensbergen and R. Minnich. Grave robbers from outer space using 9p2000 under linux. In *In Freenix Annual Conference*, pages 83–94, 2005.

[6] Latchesar Ionkov, Ron Minnich, and Andrey Mirtchovski. The xcpu cluster management framework. In *First International Workshop on Plan9*, 2006.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 conference on EuroSys*, pages 59–72. ACM Press New York, NY, USA, 2007.

[8] Kwan-Liu Ma, Chaoli Wang, Hongfeng Yu, and Anna Tikhonova. In-situ processing and visualization for ultrascale simulations. *Journal of Physics: Conference Series*, 78(1):012043+, July 2007.

[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM New York, NY, USA, 2008.

[10] R. Pike. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[11] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[12] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P.K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Symposium on Operating System Design and Implementation (OSDI), San Diego, CA, December*, pages 8–10, 2008.