Composing and Executing Parallel Data-flow Graphs with Shell Pipes

Edward Walker
Texas Advanced Computing Center
University of Texas at Austin
Austin, TX, USA
1-512-232-6579
ewalker@tacc.utexas.edu

Weijia Xu
Texas Advanced Computing Center
University of Texas at Austin
Austin, TX, USA
1-512-232-7158
xwj@tacc.utexas.edu

Vinoth Chandar
Oracle Corporation
400 Oracle Parkway
Redwood Shores, CA, USA
1-650-607-0046
vinoth.chandar@oracle.com

ABSTRACT

In this paper we extend the concept of shell pipes to incorporate forks, joins, cycles, and key-value aggregation. These extensions enable the implementation of a class of data-flow computation with strong deterministic properties, and provide a simple yet powerful coordination layer for leveraging multi-language and legacy components for large-scale parallel computation. Concretely, this paper describes the design and implementation of the language extensions in Bourne Again SHell (BASH), and examines the performance of the system using micro and macro benchmarks. The implemented system is shown to scale to thousands of processors, enabling high throughput performance for millions of processing tasks on large commodity compute clusters.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications – *data-flow languages.*

General Terms

Experimentation, Languages.

Keywords

Data-flow processing, parallel processing, coordination languages.

1. INTRODUCTION

"One of the most widely admired contributions of Unix to the culture of operating systems and command languages is the pipe." – Dennis M. Ritchie [1]

Shell pipes are a method for composing a group of applications into one with higher-level functionality. The concept was invented over 30 years ago in the original Unix shell [1], and since then, scripting with pipes has become an essential tool for many system administrators and software developers. The popularity of shell pipes can be explained by their ability to *recursively compose*: allowing for the creation of expendable software toolkits of increasing sophistication from simpler program components, which may themselves be composed from pipes. Indeed, command pipelines in the earliest Unix installations may arguably be the precursors of all modern workflows.

To date, the success of pipes has resulted in the concept being ported to a wide variety of operating systems, e.g. DOS, Windows NT, Mach, and others. The concept has also motivated implementations that have gone beyond the traditional shell interface (see section 6 for related work). However, despite its popularity, there is no prior art that has attempted to expand the syntax and semantics of Unix pipes for parallel coordination in the shell. With the proliferation of large commodity compute clusters in academia and industry, we believe it is timely to expand the concept of pipes for parallel processing. Furthermore, because the shell remains the default *login* environment to the operating system, we believe it provides an ideal coordination layer for composing multi-language software programs into higher level parallel constructs, idioms and applications -- extrapolating the original Unix idea of expandable toolkits for large distributed-memory compute clusters. Certainly, scripting in general is still an important counterpart to stronger typed system languages [2] because it remains an effective mechanism for rapidly automating common computing tasks for the non-expert and expert information technologist alike [3][4].

In this paper, we make three specific contributions. First, we extend the concept of pipes to incorporate forks, joins, and cycles, allowing the creation of data-flow graphs with similar properties to Kahn Process Network (KPN) in the operating system shell. KPNs are data-flow graphs that have strong deterministic properties, allowing for correct parallel programs to be defined without explicit locks or synchronization [5][6][8]. Second, we introduce a simple shell language extension that incorporates a higher-level concurrency pattern for key-value aggregation, allowing MapReduce [17][18] type algorithms to be constructed at the shell command line. This extension allows administrators and developers to easily construct key-value algorithms for large-scale analysis of unstructured data, similar to that proposed by other well-known systems like Hadoop [22] and Dryad [25][26], while benefitting from the deterministic properties provided by KPNs. Third, we describe an expandable implementation framework in Bourne Again SHell (BASH) that allows software

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WORKS'09, Nov. 16, 2009, Portland, OR, USA.
Copyright 2009 ACM 978-1-60558-717-2/09/11…$10.00.

agents to be plugged into the shell to modify its run-time behavior. This agent framework provides a flexible and extensible mechanism for experimenting with ever more sophisticated and robust run-time solutions for our proposed extensions in the shell. In particular, we describe our agent implementation which is shown to scale to thousands of processors, enabling high throughput performance for millions of processing tasks.

The paper is organized as follows. Section 2 introduces our proposed shell extensions by illustrating a simple parallel summation script. Section 3 then describes our pipeline semantic extensions in detail: forks, joins, cycles and key-value aggregation. Section 4 describes our design and implementation in BASH, and section 5 describes our experimental evaluation of the constructed system. Section 6 describes related work, and section 7 concludes our paper with a brief discussion of future directions.

2. A TRIVAL EXAMPLE

This section describes a simple parallel summation script in BASH using some of our proposed extensions. We first assume that a user has a list of numbers in a file called "`f.dat`". The user can compose a script to calculate the sum of the content of the file in parallel with our shell as shown in Figure 1.

```
function part()
{
  while read i; do
    key=$(($i % $N))
    # new built-in: emits key-value pair
    emit_tuple -k $key -v $i
  done
}

function sum()
{
  # new built-in: reads key-value pair
  consume_tuple -k key -v value

  num=${#value[@]}
  for i in $(seq 0 $(($num-1))) ; do
    sum=$(($sum + ${value[$i]}))
  done

  # new environment variable
  if (($_ASPECT_NUM_HASHED_KEYS > 1))
  then
    emit_tuple -k "0" -v $sum
  else
    echo $sum
  fi
}

# new cycle and key-value aggregation syntax
cat f.dat | part | (++ 2 sum on keys)
```

Figure 1. Parallel summation BASH script.

In the script the `part` function is used to partition the input data stream into N buckets, emitting key-value pairs for each datum with the key identifying the bucket to which it belongs. The key-value pairs generated by the `part` function are then piped into a cycle, iterating twice around the `sum` function. The first iteration performs N parallel partial sums for each of the buckets. The `sum` function then advertises their partial sums in key-value pairs with a common key "0". These partial sum key-value pairs are then aggregated and piped into the final `sum` task in the second iteration. This final `sum` then performs the global summation over the intermediate partial sums and prints the final result.

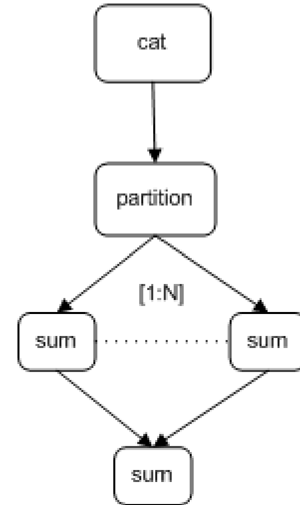


Figure 2. Parallel summation data-flow graph

The corresponding data-flow graph describing the algorithm is shown in Figure 2. The graph shows each process in the computation as nodes, with arcs indicating data-flow between the nodes.

3. SHELL PIPE EXTENSIONS

A Kahn Process Network (KPN) [5][6] is a data-flow graph where nodes represent compute processes, and arcs represent unbounded unidirectional FIFO channels with blocking read and non-blocking write semantics. In this model, compute processes with input data dependencies block on reading their input channels until all input data arrives, before performing their computation and writing their results to their outgoing channels. The compute processes are assumed to not share memory, communicate only through the channels, and have deterministic input-output behavior, i.e. the same input history in a compute process will always produce the same output history, irrespective of the process execution timing.

Given these assumptions, a KPN is shown to be entirely defined by the channel network, the compute processes, and the initial data tokens used to boot-strap the computation. Importantly, the computation in a KPN is provably deterministic [7], unaffected by the execution order of the compute processes in the network. Thus the results produced from a KPN are unaffected by the physical processors available to execute it, i.e. executing the graph on 1 processor or 10000 processors will produce the same result.

Shell pipelines describe a linear data-flow graph that approximates the properties of a KPN. The pipe operator, "|", represents the unbounded unidirectional FIFO queues connecting processing stages that block until the required input data becomes available. Current shell run-times ensure the stages in the pipeline do not block on a write by always concurrently spawning

the process in the next stage. The concurrently spawned process in the next stage is allowed to read the contents of the pipe, ensuring that the pipe is drained.

In this paper we propose to extend the semantics of shell pipes to incorporate the concept of forks, joins, cycles, and key-value aggregation, allowing KPN-type graphs to be expressed. We also implement a shell run-time that ensures the instantiated pipe channels approximate the required unbound FIFO queue semantics. The following sub-sections elaborate on the proposed language extensions, and section 4 describes the design and implementation of our run-time in detail

3.1 Spawning Parallel Tasks

To support our pipeline extensions, we introduce new semantics to allow a shell function (or command) to be executed in parallel using the syntax:

```
<A> on <n>[:<stride>][:<span>] procs
```

where `<A>` is a shell function (or command) with optional input arguments, and `<n>` is the number of tasks that needs to be spawned, irrespective of the actual number of processors available for the execution run.

Optional processor placements advice for the spawned tasks may also be provided in the `<stride>` and `<span>` terms. The `stride` term restricts the placement of tasks to the specified number of processor slots on each compute node, and the `span` term spaces the execution compute nodes to only those separated by the specified step value. This is illustrated in Figure 3.

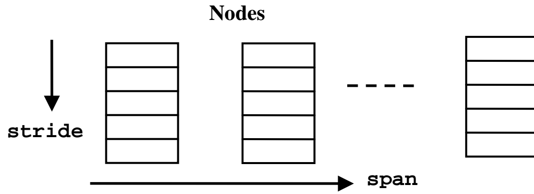


Figure 3. Task placement: stride represents processor slots per node, and span represents steps across nodes

For example, `<n>` instances of a shell function can be executed on one processor of each of the available compute nodes with the syntax:

```
<A> on <n>:1 procs
```

Or on alternatively spaced compute nodes:

```
<A> on <n>:1:2 procs
```

The default values for `<stride>` and `<span>` are the number of processor slots available on each compute node and one respectively. As a syntactical sugaring, the keyword "`all`" can be used for the `<n>` term to specify that the number of tasks to spawn equals the number of processors available in the system. As a further syntactical sugaring, the user can also specify the equivalent of the first example (`<n>` tasks on one processor on each compute node) with the syntax:

```
<A> on <n> nodes
```

Within this context, the keyword "`all`" can also be used for the `<n>` term to specify that the number of tasks to spawn equals the number of compute nodes available in the system.

3.2 Pipeline Fork

Pipeline fork is supported in our shell as shown in Figure 4. The syntax declares that the output of `<A>` be piped into "`n`" parallel instances of `<B>`, where "`n`" can take any valid task spawning specification as described in the previous section. Semantically, given a network described by the output of function A connected to the inputs of multiple instances of function B, denoted by B^1 to B^n (where A and B are deterministic), the result of the network is always $\{B^1(A(x)), \ldots, B^n(A(x))\}$, where x denotes the input data stream for A (if any).

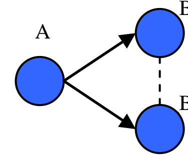
<table>
<tr><td>Graphical Representation</td><td>
</td></tr>
<tr><td>Syntax</td><td>A | B on n procs</td></tr>
<tr><td>Semantic</td><td>Output = $\{B^1(A(x)), \ldots, B^n(A(x))\}$</td></tr>
</table>

Figure 4. Pipeline fork

3.3 Pipeline Join

Pipeline join is supported in our shell as shown in Figure 5. The syntax declares that the output of "`n`" parallel instances of `<A>` be piped into `<B>`. Semantically, given a network described by the outputs of multiple instances of function A, denoted by A^1 to A^n, connected to the input of function B (where A and B are deterministic), the result of the network is always $\{B(A^1(x_1), \ldots, A^n(x_n))\}$ where x_1 to x_n denote input data streams (if any) for A^1 to A^n respectively.

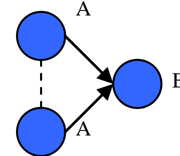
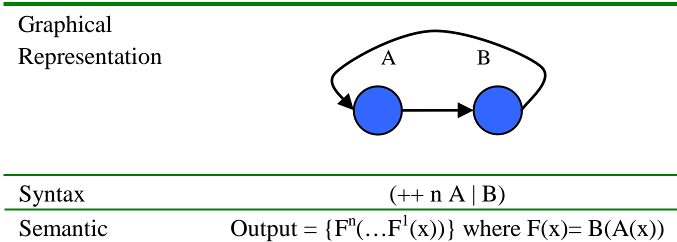
<table>
<tr><td>Graphical Representation</td><td>
</td></tr>
<tr><td>Syntax</td><td>A on n procs | B</td></tr>
<tr><td>Semantic</td><td>Output = $\{B(A^1(x_1), \ldots, A^n(x_n))\}$</td></tr>
</table>

Figure 5. Pipeline join

3.4 Pipeline Cycles

Pipeline cycles are supported in our shell as shown in Figure 6. The syntax declares a cyclic pipe, where the output of a pipeline command is piped back to its input for each of its "n" iterations. Semantically, given a network described by a deterministic function F iterating "n" times, the result of the network is always $\{F^n(\ldots F^1(x))\}$ where x denotes the initial input stream into the network, and F^i the ith instance of the function F.



Graphical Representation	
Syntax	(++ n A | B)
Semantic	Output = $\{F^n(\ldots F^1(x))\}$ where $F(x)= B(A(x))$

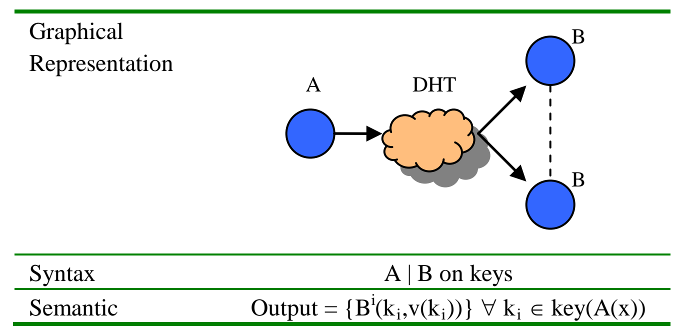
Figure 6. Pipeline cycle

3.5 Pipeline Key-Value Aggregation

Key-value aggregation is supported in our shell as shown in Figure 7. The syntax declares that an instance of <B> is spawned for each key stored in an ephemeral *distributed hash table (DHT)*, assumed to be populated with key-value pairs emitted from the previous stage of the pipeline, <A>. The key-value pairs are emitted by <A> using the built-in command `emit_tuple`, while the assigned key-value pairs can be read in <B> with the built-in command `consume_tuple`. The ephemeral DHT itself is implemented in the shell run-time and is interposed transparently between the key-value producer and consumer during the life-time of the transaction.

Semantically, given a network described by the output of A connected to the inputs of m instances of function B, denoted by B^1 to B^m, the result of the network is given by $\{B^1(k_1,v(k_1)), \ldots, B^m(k_m,v(k_m))\}$ $\forall$ $k_i \in key(A(x))$, where key(A(x)) is the set of all unique keys generated by A(x), m is the size of this set, $v(k_i)$ are the values associated with key k_i, and x is the input data stream for A.

In our key-value aggregation network, the ephemeral *DHT* process is a deterministic function because it always produces the same output stream given the same input stream. For example, the input key-value stream "(2 hi) (5 there) (2 mr) (8 wriggles)" will always produce the output stream "(2 hi mr) (5 there) (8 wriggles)". Consequently, the key-value aggregation construct preserves the deterministic properties of our KPNs.



Graphical Representation	
Syntax	A | B on keys
Semantic	Output = $\{B^i(k_i,v(k_i))\}$ $\forall$ $k_i \in key(A(x))$

Figure 7. Pipeline key-value aggregation

4. DESIGN AND IMPLEMENTATION

We have implemented our proposed pipeline extensions in version 3.2 of Bourne Again SHell (BASH) on Linux kernel 2.6.x. BASH was chosen as an initial prototype platform because of its rich set of language features compared to other shells. These language features includes exception handling, dynamic loadable built-ins, arrays, etc. However much of our design and implementation choices are agonistic of the shell itself. Hence, our extensions can similarly be implemented in other shells like Tenex C Shell, Korn Shell, and Windows PowerShell.

Figure 8 shows a high-level overview of our software architecture. In the architecture, software agents implement the parallel execution capabilities of our pipeline extensions. The software agents do this by spawning shell workers on processors in the system, assigning compute tasks to them, and serializing results from the completed tasks back to the shell. To determine what processors are available to it, the shell also integrates with cluster management systems like LSF, OpenPBS, Condor and Sun Grid Engine, querying for available compute nodes through the appropriate environment variables set by these resource managers. Finally, to facilitate fast startup of shell workers, we provide a process startup overlay to enable this. We will elucidate on all the important aspects of this implementation in the following sub-sections.

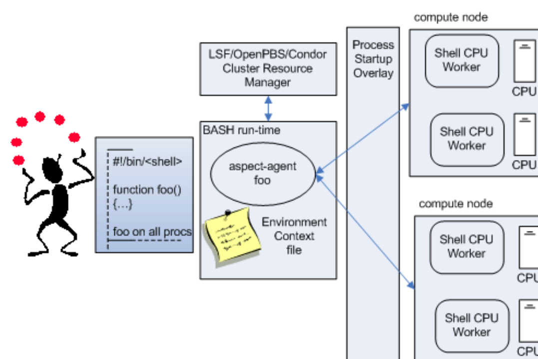


Figure 8. High-level overview of architecture

4.1 Software Agents

We use the concept of software agents heavily in our implementation. In prior work, we have described the benefits of a pluggable software agent framework that allows per-user customizable run-time behavior to be implemented in the shell [20][21]. We build on this prior work by similarly allowing external software agents to be invoked to execute a shell function or command when any of our extended execution syntax requiring parallel execution is encountered.

For example, when a user types "`foo_func on 10 procs`", the shell executes "`aspect-agent foo_func.sh`", where `foo_func.sh` is a wrapper invoking the shell function `foo_func`. An environment variable is also used to advise the software agent that 10 instances of the function is requested. It is then the responsibility of the software agent, `aspect-agent`, to perform the parallel execution of the tasks, and to serialize the results to stdout.

To ensure that function or command executes in the correct environment, our shell also generates an environment context file when the agent is invoked, and advertises this file through an environment variable. This context file contains declarations for all exported variables and function definitions in the current execution context. The software agent can then use this environment context file to ensure that all tasks have the correct environment when executed.

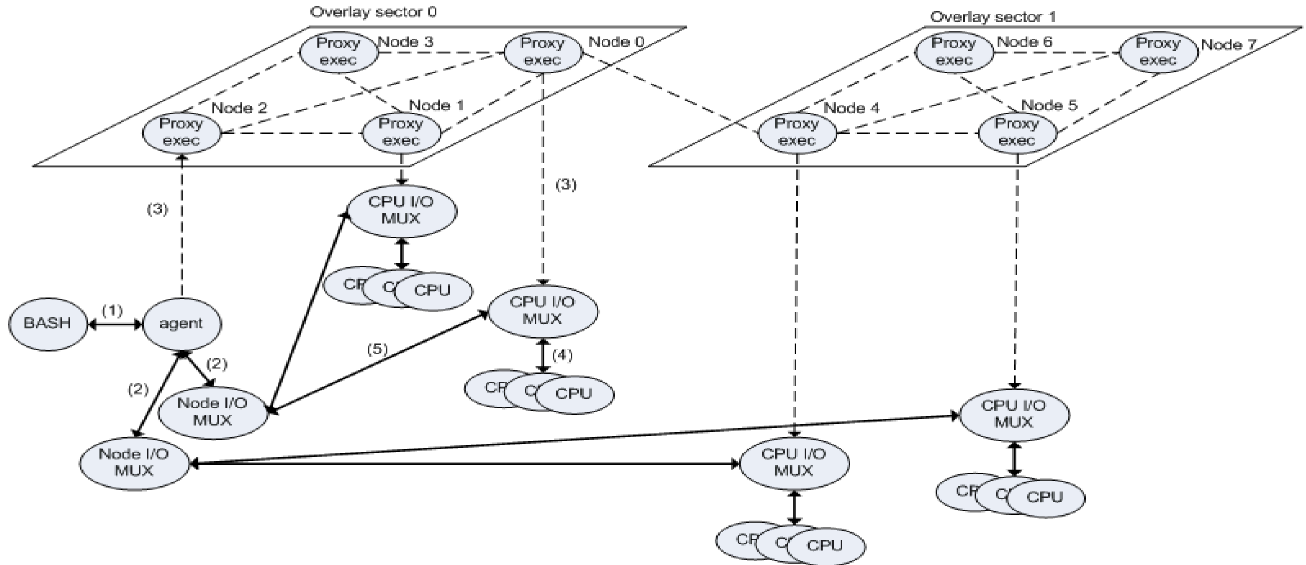


Figure 9. Software agent starting shell workers on compute nodes {0, 1, 4, 5} using the process startup overlay

4.1.1 Shell workers

Our run-time spawns shell workers for executing tasks in parallel when required. These shell workers are simply instances of our shell with the "-s" flag specified to allow commands to be issued to them through their stdin. In our execution model, when a fork, join or key-value aggregation is encountered, the run-time spawns as many shell workers as possible: each worker assigned to a CPU processor, up to the number of available CPU processors in the cluster. The tasks involved in the fork, join or key-value aggregation are then streamed through these shell workers to enact the required semantics.

An illustration of the sequence of actions involved in starting shell workers across processors in the system is shown in Figure 9. The steps are as follows:

Step 1: A software agent is spawned to execute a command when a fork, join or key-value aggregation is encountered in the shell.

Step 2: The agent forks a number of node IO multiplexer processes. These node IO multiplexer processes are responsible for routing messages to/from subsets of nodes used in the parallel execution, and are used to avoid exhausting the per-process socket descriptor limit.

Steps 3: The agent then invokes the process startup overlay to execute a CPU IO multiplexer process on each node involved in the parallel execution.

Step 4: The CPU IO multiplexer process starts shell workers for each processor on the compute node, and is responsible for routing messages to/from the shell worker started.

Step 5: The CPU IO multiplexer makes a callback to the node IO multiplexer to establish a network connection between the software agent and to each of the shell workers spawned.

The shell workers have their stdin and stdout duplicated (`dup`) to sockets connected to the network path to the software agent. This is the primary access mechanism in our infrastructure for the agents to issue command workloads, and examine output results, to/from the shell workers. In addition to the stdin and stdout, a control channel is also created and duplicated to a socket connected to the network path to the agent. The write and read ends of the channel are advertised in the environment, and used by the `emit_tuple` and `consume_tuple` built-in commands for sending and receiving key-value pairs respectively.

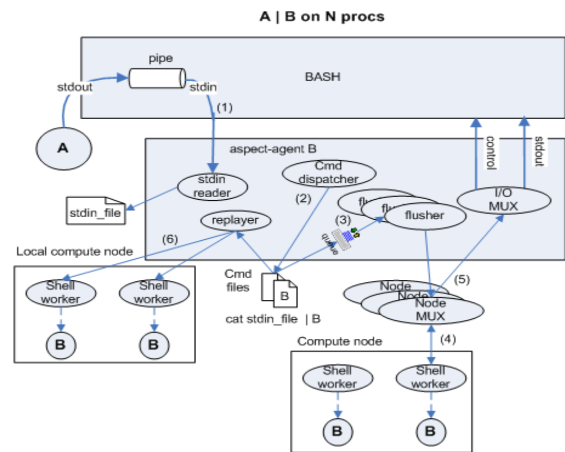


Figure 10. Implementation of pipeline fork

4.1.2 Remote tasks and pipes

Figure 10 details the steps implemented by our software agent to support the pipeline join construct. The steps for implementing the pipeline fork construct are similar.

In our shell, when a pipeline command "`A | B on N procs`" is executed, the run-time concurrently forks-execs the commands "`A`" and "`aspect-agent B`", setting up a pipe between the output of "`A`" and the input of "`aspect-agent B`". The following steps then occur:

Step 1: The software agent reads the output from "A" and stores this in a temporary file, `stdin_file`. This approximates the unbounded FIFO semantics in KPNs.

Step 2: The agent then composes a command file that sources the environment context file, executes the command string "`cat stdin_file | B`", and removes itself on completion (i.e. "`rm -f $0`"). As an optimization, for the case where *N* is greater then the number of available processors in the cluster (*M*), this command string is also wrapped in a loop with *N/M* iterations. When this optimization is performed, a restart file is also created for logging the last iteration executed to facilitate the replay of the command file on node failure.

Step 3: Once the command file is composed, a command dispatcher thread pushes execution requests for this command file into FIFO queues associated with a set of command flusher threads. In the agent, a command flusher thread is created for each node IO multiplexer in the current execution, whose sole purpose is to detect enqueued command requests and to send them, in a round-robin order, to the shell workers. After all command files are enqueued, the command dispatcher issues a final "exit" command to all the shell workers.

Steps 4/5: Commands are streamed to the shell workers by the flusher threads, and the output (stdin, stdout, and control) of the workers routed back to an IO multiplexer thread in the software agent. This IO multiplexer thread is responsible for serializing the messages it receives and forwarding them to the stdin, stdout and control descriptors in the BASH run-time. The IO multiplexer is also responsible for detecting when shell workers exit, and signals to the agent when all workers terminate.

Step 6: Before exiting, a replayer thread in the software agent checks if all command files have successfully executed, and hence removed. If command files exist, local shell workers are spawned, one for each CPU processor available on the node, and the orphaned command files re-dispatched to these local shell workers. The software agent terminates after all the orphaned command files complete successfully.

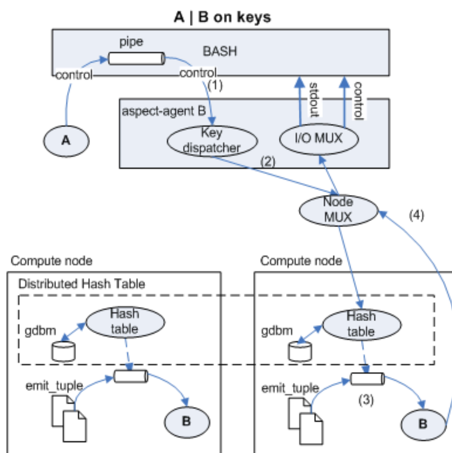


Figure 11. Implementation of key-value aggregation

4.1.3 Key-value aggregation and distributed hash-tables

Figure 11 shows the steps taken by our agent in implementing key-value aggregation. When the pipeline command "`A | B on keys`" is executed, the run-time concurrently forks-execs the commands "A" and "`aspect-agent B`", setting up a pipe between the output of "A", and the input of "`aspect-agent B`". In addition, a pipe is also created for a control channel, with the read-write ends advertised in environment variables for use by the built-in commands `emit_tuple` and `consume_tuple`. The following steps then occur:

Step 1: Command "A" advertises key-value pairs through the control channel with the `emit_tuple` command, and the software agent reads the advertised key-value pairs in a key dispatcher thread.

Step 2: The key dispatcher forwards the key-value pair to a compute node determined by *hash(key)%M*, where *hash* is a hash function that generates a 32-bit integer for a given key string, and M is the number of compute nodes in the system. The key-value pair is then stored in an associative array data structure on the designated compute node. This associative array is implemented as a hash table in memory, and offloaded to a GDBM file-based hash table if the table size exceeds an environment configurable threshold.

Step 3: When all key-value pairs have been dispatched, the key dispatcher sends a special EOF header to the compute nodes participating in the distributed hash table. Each compute node then generates a binary file, `tuple_file`, for each key in its associative array with its aggregated values, and composes a command file for execution by workers spawned on each processor on the local node. This command file is constructed to source its environment context file, execute the command string "`emit_tuple -f tuple_file | B`", and remove itself on completion (i.e. "`rm -f $0`"). The –f option flag used with `emit_tuple` simply tells the command to emit the key-value data stored in the specified file. As an optimization, for the case where the number of key aggregated (*N*) on the local node is greater then the number of available processors on the node (*M*), this command string is also wrapped in a loop with *N/M* iterations. When this optimization is performed, a restart file is also created for logging the last iteration executed to facilitate the replay of the command file on node failure.

Step 4: Finally, the output (stdin, stdout, and control) of the executed commands are routed back to an IO multiplexer thread in the software agent. This IO multiplexer thread is responsible for serializing the messages it receives and forwarding them back to the stdin, stdout and control descriptors in the BASH run-time.

4.2 Pipeline Cycle Implementation

The implementation of the pipeline cycles construct in BASH is straightforward, involving less than 100 lines of code change. In our implementation, the parser was updated to recognize the pipeline cycle syntax and to set a flag in the shell's command pipeline data structure. The run-time then checks for this flag when executing a pipeline command, wrapping a loop around the command if the flag is detected. The loop is responsible for copying the output of the last command into a file, and piping the content of that file back to the first command on subsequent

iterations. This is achieved by creating two additional virtual pipes, and spawning reader and writer threads responsible for reading the output at the end of the pipeline into a file, and writing the content of the file back to the beginning of the pipeline respectively.

4.3 Process Startup Overlay

When a user starts our shell as a login environment, or runs a script with our shell, the shell run-time boot-straps a process startup overlay across each of the compute nodes available to the user. To integrate with cluster resource managers, the list of available nodes to the shell run-time can be set in a batch job script to the nodes allocated by a resource manager for the particular job run. Our startup overlay then serves two purposes: it enables fast startup of shell workers (and the functions/commands they execute), and it endeavors to maintain continuous operation of the script in the presence of compute node failures.

At a high-level, our startup overlay provides a two-level routing hierarchy for forwarding commands between nodes in our network. It is organized as a set of fully connected *sectors*; with each sector containing a set of fully-connected *proxies* each representing a physical compute node. The connections between sectors are routed through a single proxy in each sector, called the *sector head*.

To establish this routing hierarchy, each compute node is assigned a monotonic increasing 32-bit node ID, of which the first X bits designate its proxy ID (2-bits are used in Figure 9 for illustration and 8-bits are used in the actual implementation), and the remaining bits designate its sector ID. In our overlay, starting a command within the same sector incurs one routing step. While starting a command between sectors incurs at most three routing steps: the command is forwarded through the local sector head to its peer sector head, before finally arriving at the intended destination proxy.

Our startup overlay informs the software agents in our framework of available nodes in a `master_node` file, maintained by an elected primary sector head and exported in an environment variable. Fault-tolerance is provided by the startup overlay because the sector heads continually monitors the health of the proxy connections within their own sector and the connection between sectors. If a sector head detects a node failure, it updates the primary sector head with this information. The primary sector head is then responsible for updating the `master_node` file to exclude the faulty nodes. Subsequent invocations of our software agent will then use this updated `master_node` file, containing only nodes with which the overlay has routable connections.

5. Experimental Evaluation

We examine the performance of our system in a number of ways. First, we use micro benchmarks to examine 1) the effectiveness of our process startup overlay, and 2) the performance of executing millions of task and key-value aggregations. Second, we look at the performance of the system in implementing a solution for the TeraSort benchmark [29].

In this section, all benchmarks were run on a production scientific compute cluster managed by the University of Texas at Austin. The cluster consists of 4000 compute nodes, each composed of four AMD 64-bit Barcelona Quad-Core processors (16-processor cores), with 32 GB of memory, and 300 MB of disk storage each. All compute nodes are interconnected by a high-speed InfiniBand switch, and a Lustre parallel file system is available for global file sharing.

The cluster is a multi-user system, shared by a large community of computational science researchers. Therefore access to the cluster is space and time shared through a batch queuing system provided through the SGE cluster resource manager.

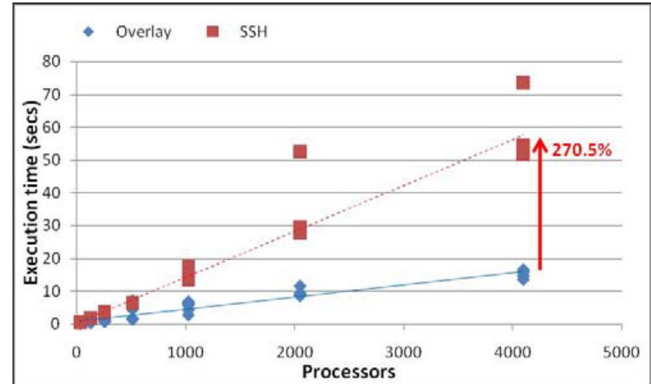


Figure 12. Startup performance of a simple command with the overlay and the default SSH mechanism

5.1 Command Startup Performance

The default remote command startup mechanism on the cluster is the SSH command. We therefore compare the performance of executing a simple command in parallel on processors in the cluster using the default mechanism and our startup overlay. Figure 12 shows the times to complete the execution of the `hostname` commands on 32 to 4096 processors with the two mechanisms. For the startup overlay case, we ran the command "`hostname on all procs`" in a shell script we submitted as a batch job to the cluster. For the experiment we record up to six runs for each benchmark, as the variance in the command startup performance using SSH was observed to be large. A linear interpolated fit of the data was then used to observe the trend in the startup performance of both mechanisms.

Our results show that the performance of executing commands in parallel in our overlay is much better than using SSH. This is especially evident for large processor runs. At 4096 processors, we see that the command execution performance is approximately 270% worse using SSH than when our overlay is used.

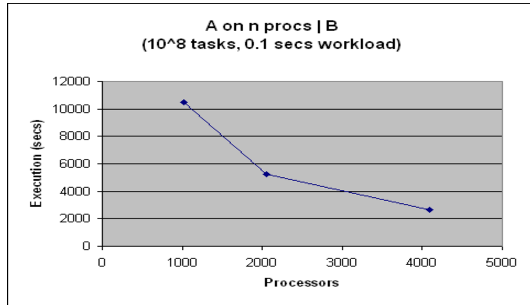
The SSH command is commonly used as the default remote execution mechanism for many commodity clusters. For example all the parallel clusters at the NSF supercomputing centers use SSH for this purpose. However, our process startup overlay uses the SSH mechanism only to boot-strap across the participating compute nodes at the beginning of job execution. After the overlay is established, all remote commands in our system incur the benefit of the exhibited low overhead.

5.2 Task Execution Performance

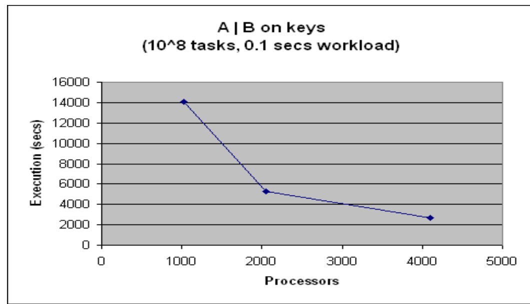
In this experiment, we observe the performance of a pipeline join (`A on n procs | B`), involving 10^8 tasks, and key-value aggregation (`A | B on keys`), involving 10^8 unique key-value

pairs. In this syntactic benchmark, we assume that task A in the pipeline join, and task B in the key-value aggregation consume 0.1 seconds of work, emulated by the command "`sleep 0.1s`".

Figure 13 shows the execution times of the respective experiments for runs involving 512 to 4096 processors on the compute cluster. We observe that the performance is approximately similar, and improves as the processor set size increases.



(a)



(b)

Figure 13. Performance of (a) pipeline join and (b) key-value aggregation for 100 million tasks

We also observed the key distribution of our ephemeral DHT in the above key-value aggregation experiment. Figure 14 shows the distribution of reduce tasks across the 256 compute nodes (for the 4096 processor experiment) associated with the 100 million generated keys. We calculate the mean (X) and standard deviation (σ) for the key distribution to be 390625 and 664 respectively, indicating a good uniform distribution (low variance) across the compute nodes in the DHT of our system. The results are unsurprising but important in verifying our implementation.

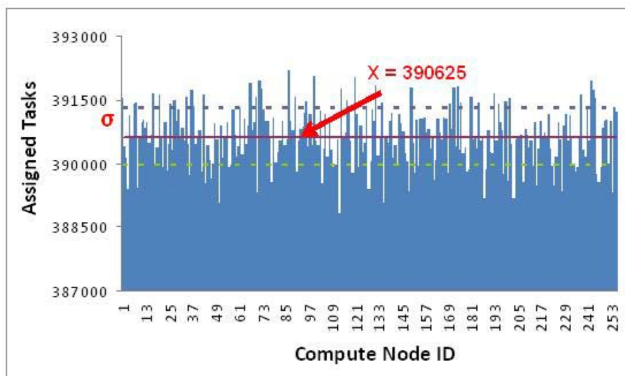


Figure 14. Distribution of reduce tasks over compute nodes

5.3 TeraSort Benchmark

We also examine a solution for the TeraSort benchmark [29] using our shell. The task is to sort 100-byte data records generated from an unmodified `gensort` executable.

Our solution script consists of three phases. First, we spawn the data generator program in parallel on each processor socket (four instances on each compute node). A `LD_PRELOAD` shared object then transparently partitions the data generated from each `gensort` instance by intercepting calls to `fwrite()`, and permitting writes to local disk if the first two bytes (16-bits) of the record falls within the range

$$\left[2^{16} * \frac{T}{N}, \qquad 2^{16} * \frac{T+1}{N}\right]$$

where T is the task identifier and N is the total tasks involved in the computation.

Second, we spawn a task on each processor socket to perform the actual sort on each of these range partitioned files on local disk. Third, we spawn a task on each compute node to copy the sorted data from the local disk into the global file system. Thus, our solution represents a version of parallel bucket sort[1].

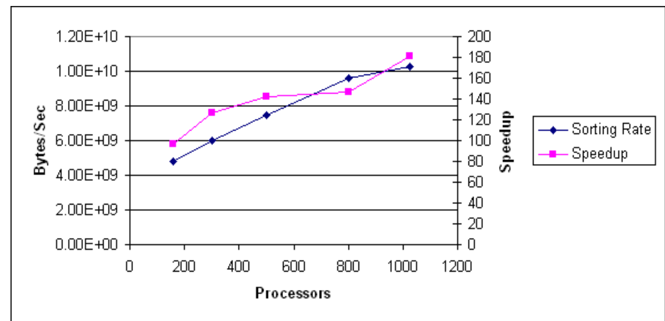


Figure 15. Sorting rate and speedup shown with increasing number of processors. Data size = 6e10^{10} bytes

Figure 15 shows the sorting rate, and the speedup, achieved for increasing processor set sizes involved in the computation. For each processor set size, N, we sort a total of N*6e10^{7} bytes. The solution with our shell shows close to linear speedup for increasing processor set sizes. Importantly, our shell enabled us to derive a parallel solution for this problem on a production scientific cluster within a few hours of scripting and testing effort.

5.4 Other Applications

We have also implemented a number of additional applications using our shell on the compute clusters managed by the Texas Advanced Computing Center. These applications include word-count, parallel grep, matrix-multiply and K-means clustering. More information about these applications, including their source code, can be found on the project web site [31].

---

[1] The source code for our implementation can be found on the project web site[31].

6. RELATED WORK

The important significance of coordination languages have been extensively described in the literature [9][10]. A coordination language project that closely relates to our work is Ptolemy [7][8][11]. The project is developing a suite of languages, called domains, for the design and simulation of concurrent, real-time, embedded systems. In particular, the PN (Process Network) domain most closely resembles our work as it can also be used to implement KPN graphs. However, the project differs from our work in their focus on real-time embedded systems, while we focus on large scale distributed computation. Also, their programming environment is primarily the graphical program editor, Vergil, while we use commodity operating system shells as the program composition framework.

The pipe concept has also been widely adopted by many other projects. Karges et. al. [34] described an early design of a parallel pipe implementation, but their work only touches on the fork semantics introduced in this paper. CMS pipelines [12], in VM/ESA and z/VM, allow the creation of multiple input and output channels in a pipeline command. A Python version of CMS pipelines is also available [13], providing this feature to other operating systems. However, CMS pipelines are different from our proposal because they do not support cycles, parallel processing, or higher level concurrency patterns like MapReduce.

Object pipes have recently been introduced [14], most notably in Windows PowerShell [15]. Our implementation in BASH does not yet support object streams. However, our proposed extensions do not preclude their introduction into shells that support this concept in the future.

Web products like Yahoo!Pipes [16] have also been introduced recently. The product allows web content to be filtered and modified by connecting RSS feeds through pipe-like networks. However the concept does not support parallel computation.

Systems for coordinating MapReduce programs are also closely related to our work. Phoenix [19] provides a low level C/C++ API library for implementing MapReduce on shared memory architectures. In addition, open source projects like Apache Hadoop [22] allow MapReduce to be implemented through a Java API. Our approach implements MapReduce at the coordination language layer, allowing arbitrary programs, including legacy programs, to leverage MapReduce concurrency patterns, on shared-memory and distributed memory architectures. Most importantly, compared to these other systems, we provide a more general computational model in KPN for implementing a wider range of algorithms.

Dryad [26], and its associated programming interfaces [25][28], is a general purpose distributed execution engine for direct acyclic (DAG) data-flow computation, and hence is very closely related to our system. However, our work differs with Dryad in three important respects. First, Dryad only supports acyclic graphs, while our system supports graphs with cycles. Second, Dyrad does not integrate with commonly used cluster job managers like OpenPBS, SGE, etc. (Dryad is intended to be the cluster job manager itself), while our system is designed for use on clusters using commodity job managers. Third, Dyrad has not yet been integrated with commodity operating system shells like BASH, which our work specifically addresses.

Other proposed systems that also use the concept of pipes for parallel computation include pipelets [27], but they use the XML Pipeline Definition Language (XPDL) for describing the relationship between components in a pipeline.

Finally, our work is also related to functional and declarative languages for parallel processing like Sawzall [23], Pig Latin [24] SCOPE [28], and Orc [33]. Our work differs from these because we do not propose a new language interface but instead extend an existing commodity shell scripting interface for parallel processing with pipes.

7. CONCLUSIONS AND FUTURE WORK

We have proposed semantic extensions to the concept of shell pipes, verified the utility of our proposal by implementing a prototype in BASH, and composed large applications using the system on a production distributed-memory compute cluster. In the process, we are learning some interesting lessons.

First, we were made very aware that debugging tools are critically needed for shells in a parallel computing environment. There are currently few tools which can be used to investigate anomalous behaviors in our pipelines. When things go wrong, as they sometimes do, we had to examine the output of each individual stage manually to debug the problem. Therefore we wish to investigate debugging tools in the future, possibly extending the built-in BASH debugger (`bashdb`) for our purpose.

Second, we were reminded that the actual performance of a parallel shell script is determined by many additional run-time factors, such as process startup overheads, file system bottlenecks, L1/L2 memory caching effects, etc. We are interested in investigating simulators that can be run on a workstation to help us predict the performance of our parallelization choices. This will greatly aid users in composing first-time right scripts, preventing unforeseen performance surprises when the scripts are actually run on production parallel computers.

Third, we are still unclear how to proceed with the integration of our pipeline extensions with named pipes (`mkfifo`). Future releases of our BASH shell may incorporate this support, but as yet, it remains an open question.

8. REFERENCES

[1] Dennis. M. Ritchie, "The Evolution of the Unix Time-Sharing System", *AT&T Laboratories Technical Journal*, vol. 6(2) (Oct. 1984), 1577-1593, Oct. 1984.

[2] R. P. Lai, "In Praise of Scripting: Real Programming Pragmatism", *IEEE Computer*, 22-26, July 2008.

[3] L. Prechelt, "An Empirical Comparison of Seven Programming Languages", *IEEE Computer*, vol. 33(10), 23-29, Oct. 2000.

[4] J. K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century", *IEEE Computer*, 23-30, Mar. 1998.

[5] G. Kahn, "The Semantics of a Simple Language for Parallel Programming", *Information Processing*, vol. 4, 471-475, 1974.

[6] G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes", *Information Processing*, vol. 7, 993-998, 1997.

[7] T. M. Parks, Bounded Scheduling of Process Networks, PhD Thesis, University of California at Berkeley, 1995.

[8] Edward Lee, "Dataflow Process Networks", *Proceedings of IEEE*, vol. 83(5) (May 1995), 773-801, 1995.

[9] D. Gelernter and N. Carriero, "Coordination Languages and their Significance", Comm. ACM, 35(2), 97-107, 1992.

[10] G. Papadopoulos, and F. Arbab, Coordination Models and Languages, *Advances in Computers – The Engineering of Large Systems*, vol. 46, Academic Press, 329-400, 1998.

[11] Edward Lee et.al. The Ptolemy Project. (online) http://ptolemy.eecs.berkeley.edu/

[12] J. P. Hartmann. 2007. CMS Pipelines Explained. (online) http://vm.marist.edu/~pipeline/pipjarg.pdf

[13] B. Gailer. Python-pipelines: A Python Implementation of Hartmann (CMS) Pipelines. (online) http://code.google.com/p/python-pipelines/

[14] S. Macdonald, "Rethinking the Pipeline as Object-Oriented States with Transformations", *Proc. 9th International Workshop on High-Level Programming Models and Supportive Environments (HIPS'2004)*, 12-21, 2004.

[15] D. Jones, "Windows PowerShell: Rethinking the Pipeline", *Microsoft TechNet Magazine*, July 2007.

[16] Yahoo!Pipes. (online) http://pipes.yahoo.com/pipes/

[17] J. Dean, and S. Ghemawat, "MapReduce: Simplified Data Processing for Large Clusters", in *Proc. of 6th Symposium on Operating System Design and Implementation (OSDI'04)*, (2004), 137-150, 2004.

[18] H. Chih Yang, A. Dasdan, R-L. Hsiao, and D. S. Parker, "Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters", in *Proc. of SIGMOD International Conf. on Management of Data*, 2007.

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

[20] E. Walker, and T. Minyard, and J. Boisseau, "GridShell: A Login Shell for Orchestrating and Coordinating Applications in a Grid Enabled Environment", *Proc. of International Conference of Computing, Communications and Control Technologies*, 182-187, Austin, TX, August 2004.

[21] ApectShell: Aspect-oriented scripting shells. (online) http://www.tacc.utexas.edu/~ewalker/gridshell/GridShell.htm

[22] Apache Hadoop. (online) http://hadoop.apache.org/core/

[23] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall", Scientific Programming 13(4), 2005.

[24] C. Olston, B. Reed, U. Srivastave, R. Kumar, and A. Tomkins, "Pig Latin: A not so Foreign Language for Data Processing", in *Proc. of International Conf. on Management of Data (Industrial Track)*, 2008.

[25] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language", in *Proc. of 8th Symposium on Operating System Design and Implementation (OSDI'08)*, 2008.

[26] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks", in *Proc. of European Conference on Computer Systems (EuroSys)*, 2007.

[27] J. Carnahan, and D. DeCoste, "Pipelets: A Framework for Distributed Computation", *W4: Learning in Web Search*, 2005.

[28] R. Chaiken, B. Jenkins, P-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets", in *Proc. of International Conference of Very Large Data Bases (VLDB)*, 2008.

[29] TeraSort Benchmark. (online) http://www.hpl.hp.com/hosted/sortbenchmark/

[30] Project Gutenberg. (online) http://www.gutenberg.org

[31] Dataflow Graphs in System Shells. (online) http://sites.google.com/site/ewalker544/dataflowshell

[32] TOP500 Supercomputing Sites (online), http://www.top500.org.

[33] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra, "The Orc Programming Language", in *Proc. FMOODS/FORTE*, Springer Verlag, LNCS 5522, 1—25, 2009

[34] J. Karges, O. Ritter, and S. Suhai, "Design and Implementation of a Parallel Pipe", *Operating System Reviews*, 31(2), 64-94, 1997.