Comparative Studies Simplified in GPFlow

Lawrence Buckingham, James M. Hogan, Paul Roe, Jiro Sumitomo, and Michael Towsey

Queensland University of Technology
GPO Box 2434, Brisbane QLD 4001. Australia
{l.buckingham,j.hogan,p.roe,j2.sumitomo,m.towsey}@qut.edu.au

Abstract. We present a novel, web-accessible scientific workflow system which makes large-scale comparative studies accessible without programming or excessive configuration requirements. GPFlow allows a workflow defined on single input values to be automatically lifted to operate over collections of input values and supports the formation and processing of collections of values without the need for explicit iteration constructs. We introduce a new model for collection processing based on key aggregation and slicing which guarantees processing integrity and facilitates automatic association of inputs, allowing scientific users to manage the combinatorial explosion of data values inherent in large scale comparative studies. The approach is demonstrated using a core task from comparative genomics, and builds upon our previous work in supporting combined interactive and batch operation, through a lightweight web-based user interface.

Keywords: Comparative Studies, Collection Processing, eScience.

1 Introduction

Collection of large quantities of data is perhaps the defining characteristic of eScience, one especially noticeable in the earth and biological sciences through high throughput remote sensing and sequencing machines. Collected data is typically stored in a database and subsequently analyzed and visualized to address particular scientific questions or discover new knowledge. In the latter case, data, as opposed to theory or experiment, is starting to drive science.

The quantities of data now being collected preclude manual analysis, and a computational approach has been inherent from the start. In consequence, much of the 'low hanging fruit' from these data sets is now routinely picked, and attention has turned to more sophisticated questions and less obvious signals, often based on fine-grained comparison across large sets of data records. Such comparative studies are essential to the new science, and scientific workflow systems must evolve to support this new reality through: (i) integrated collection and input association processing; (ii) mechanisms for result traceback and ensuring processing integrity; and (iii) tools for result filtering and aggregation. This paper is motivated by large-scale comparative genomics, but the work is applicable across many disciplines, with the management of input collections and their combinatorial output values a common theme.

M. Bubak et al. (Eds.): ICCS 2008, Part III, LNCS 5103, pp. 491–500, 2008.

Bioinformatic sequence data is typically analyzed through a pipeline of different tools, perhaps to align sequences and search for motifs. Tool pipelines are either realized manually or through some kind of script or workflow system. The explosive increase in the number of genomes available has made single sequence analyses almost obsolete. Bioinformaticians now wish to compare and analyze multiple versions of similar sequences, and the greater statistical significance afforded by automated comparisons is vital to scientific investigation.

Unfortunately, existing automation tools make such studies difficult; typically they require some level of programming and provide limited support for experimentation. Moreover, responsibility for managing the combination and selection of constituent data streams remains largely with the user, as does the task of ensuring traceable associations between input data tuples and result values. The next generation of workflow systems should support large scale collection processing in a manner transparent to the scientific user. Such a user should be able to analyze a single data point, e.g. a sequence, to automatically lift the analysis to operate across a set of data points, and subsequently to apply synthesis operations to the resulting collection of values. Few available systems transparently support both these properties without requiring an element of programmer intervention. Interactive experimentation in comparative studies is poorly supported. An interactive map-reduce [1] workflow paradigm is required.

In this paper we present a novel workflow system for undertaking comparative studies, which supports interactive experimentation, automatic lifting to collection oriented computation, and automatic input association and synthesis of collections. This work extends our previously reported GPFlow system [2] – which leverages a commercial workflow system – by carefully applying the principles of structure data flow to produce a system focused on comparative studies.

The next section describes approaches taken to collection oriented computation within scientific workflow, and describes several other systems which support comparative studies through this approach. Sec. 3 presents our model for collection oriented workflow, with an illustration of the approach through the phylogenetic tree inference workflow following in Sec. 4. We conclude by examining the future directions for this work and its applications.

2 Background and Related Work

Numerous workflow systems have now been developed to support scientific research. Most have excellent support for integrating legacy tools, data and web services, but few cater for experimentation or provide an elegant model for comparative studies, if they directly support comparative studies at all. Fox and Gannon [3] observe that scientific workflow is strongly influenced by earlier work in the areas of data flow programming and distributed parallel programming. In this section we focus on collection processing in a data flow context. To that end, we first review models of data flow programming that relate to collection processing. We subsequently examine collection processing in two extant scientific workflow systems, namely Kepler [4], [5] and Taverna [6].

2.1 Data Flow Programming Models

In a data flow programming environment [7] a program is considered to be a directed graph in which nodes represent instructions while edges represent data transfer connections. Data arrives at a node on incoming edges (input ports) and leaves via outgoing edges (output ports). A node becomes eligible to fire when a value is available at each of its input ports. When a node fires, it removes values from the input ports, performs a computation, and places the resulting values in its output ports.

Most existing data flow programming environments are based on the token stream model. In this model, values are carried along the edges of the graph by tokens. Nodes are stream processors, and edges are realized by queues of tokens. When a node fires, it removes one token from each incoming queue and generates a new token for each output port. At each output port, which may be the origin of one or more edges, a distinct copy of the resulting token is appended to each queue connected to the port.

Since collection processing is fundamentally an exercise in the structuring of data, it is salutary to examine data structures in data flow environments. Davis and Keller [8] proposed two approaches that might be taken to implement data structures. The token-based tuple model introduces tuple-type tokens, which in turn contain other tokens, including nested tuple tokens where necessary. A tuple would correspond to an array in a procedural language, with elements accessible by index selection.

An alternative to the token stream model is also described in [8]. In the structure model, a data structure is incrementally constructed on each edge of the data flow graph. Several interpretations would be available for this structure, including that of a token stream, a tuple, a tree or a scalar value. During a computation, the entire history is retained in the form of the accumulated collection of data structures. In traditional data flow research this model has found little support, chiefly due to the potentially burdensome memory requirement. However, in scientific workflow often as not the history of a computation is as important as the final outcome. With the shift away from data flow at the microprogramming level, where the entire history needed to remain within physical memory, to coarse grain data flow with ready access to massive external storage, this is a less prohibitive consideration.

2.2 Collection Processing in Scientific Workflow Systems

In this section we describe collection processing in two widely used scientific workflow systems, Kepler [3], [5] and Taverna [6]. We summarize by positioning collection processing in GPFlow relative to these two systems and the overarching data flow theory.

Kepler is an extension of Ptolemy II [9], a mature and powerful visual data flow programming system in which nodes are called actors, edges are called channels, and actors communicate by sending messages along channels, following the token stream data flow model. Ptolemy II exhibits many features to be found in a general purpose programming language: conditional execution, iteration and procedural abstraction. Moreover, Ptolemy II provides a broad range of data flow execution modes.

Kepler implements collection processing by extending the component library rather than modifying the underlying language. A system that allows for hierarchical structures to be encoded and streamed through the actor pipeline is described in detail in

[3]. The presence of a collection is indicated by special delimiter tokens which mark the beginning and end of a collection. Collections may be nested arbitrarily using balanced begin-end tokens. Collection-aware actors are able to detect the delimiter tokens and act on them in a manner akin to SAX XML parsing. Collection-aware composite actors introduced in [5] allow conventional actors to be encapsulated and lifted to a "collection-aware" mode. The collection-aware composite actor uses scope expressions to characterize those parts of the token stream that are of interest and to specify how they are to be processed, including explicit support for iteration.

Taverna [6] is the scientific workflow management tool for the myGrid project [10]. Workflows are internally specified using the Simple conceptual unified flow language (Scufl), and enacted by the Execution Flow layer, which provides limited collection processing. Taverna provides a light-weight internal object model that permits the representation of lists and trees and the attachment of mime types to objects flowing through the system, exhibiting some properties of the structure model of data flow. Taverna insulates the user from explicit computational detail by providing implicit data-driven iteration over collections. A component, designed to read and write single values, is able to be lifted in a map-like way and applied to each element of a list, producing a list as the result. However, processing is less convenient if a component receives two or more lists in place of the singleton input values it was designed to accept. Here, user intervention is required to determine the association mode between the lists. Moreover, processing is constrained to operate over either the cross or dot product of the two lists.

Philosophically, GPFlow lies closer to Taverna, shielding users from computational aspects of workflow and freeing them to concentrate on their data and the scientific question at hand. However, in GPFlow collection processing is an integral facet of the execution regime, based on a variant of the structure data flow model. As in Taverna, a map-like operation lifts the workflow to operate over sets of input values, but GPFlow uses the data flow graph to determine automatically the association mode to be employed for converging set-valued input streams, and introduces a novel key aggregation and slicing mechanism to facilitate selection and ensure processing integrity. Collection processing is further enhanced by providing a simple operation that allows the user to gather disparate values to form an array for subsequent processing. Our model is described in the following section.

3 Collection Processing in GPFlow

GPFlow provides an interactive web-based workflow environment which allows the user to construct workflows from scientifically meaningful components without programming. The system implements a structured data flow model, in which a cumulative data structure is created over the lifetime of a computation. A GPFlow workflow presents as an acyclic data flow graph, yet provides powerful iteration and collection formation capabilities. The remainder of this section describes collection processing in GPFlow, in three stages: a conceptual data model; the iteration model supported by GPFlow; and finally the rules used to create collections from disparate data items.

3.1 Implementing the Data Model

A step in a scientific workflow is the execution of a computational tool. In GPFlow, these tools are wrapped and exposed to users as *Components* and a GPFlow workflow is the execution of a sequence of these components. This sequence is represented by organizing components as vertices in an acyclic data flow graph. The use of acyclic graphs enables the entire execution history of a workflow to be captured using only a simple data structure. In the graph, the termination of an edge at a particular node represent component inputs, and the origin of edges represent component outputs. We refer to the binding sites of edges as *channels*.

If a component takes input parameters, then the value of each parameter can be either be *bound* to the output of another component, shown as a directed edge between nodes, or be *unbound* and provided directly by the user through the user interface.

If the user specifies a collection through the UI, or binds a collection of values to a component's input channel, then the component will be executed for each element in the collection. Each execution is represented by a *Processor*, which encapsulates the result of the execution as well as the input value used to obtain the result. Collectively, the set of processors form the component's *result set*.

The ability to bind collections to channels means that each component output channel implicitly defines an output collection, which consists of the set of corresponding outputs from the Processors in the result set. When a component's input channel is bound to the output channel of another component, and that output channel produces a collection, then that component in turn will iterate over the collection to itself produce collection of outputs.

3.2 Data Driven Iteration

At the most basic level, a Component iterates over the Cartesian product of its input collections and queues one Processor for each combination. This works well if the data flow graph is a simple pipeline or strictly divergent tree structure, but if the data flow graph contains a fork-merge sub-graph, the simple iteration model breaks down by introducing spurious computations that could never have occurred if the user inputs were each entered in manually. Consider the workflow illustrated in Fig. 1.

Fig. 1 depicts a hypothetical workflow containing three components, each representing a step in a user's workflow. These are denoted by boxes labelled *C1…C3*.
C1 is set to iterate over a collection of user supplied inputs, represented by the multi-document icon labelled *U*. Result sets generated by components in the workflow are represented by multi-document icons labelled *A, B, C*. Here, component 1 has no bound input channels. Component 2 has an input channel bound to the sole output channel of component 1. Component 3 has two input channels, bound to the outputs of components 1 and 2.

If the input collection U, contained only one element, $u_1 \mid u_1 \in U$ i.e. $U=\{u_1\}$ then the workflow would execute as follows:

1. Step 1 executes f_1, consuming the single supplied value u_1 and producing a single result $a_1 = f_1(u_1)$, where $a_1 \in A$, $A=\{a_1\}$
2. Step 2 executes f_2, consuming the single supplied output value o_2 and producing a single result $b_1 = f_2(a_1)$, where $b_1 \in B$, $B=\{b_1\}$
3. Finally, step 3 executes f_3, producing $c_1 = f_3(a_1,b_1)$, where $c_1 \in C$, $C=\{c_1\}$

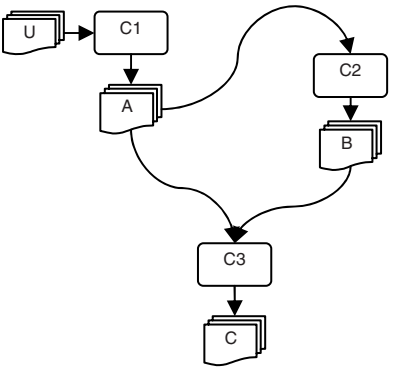

Fig. 1. Fork-merge sub-graph

Note that the cardinality of A, B and C is 1 when the cardinality of U is 1. On the other hand, if U contained multiple elements, data-driven iteration lifts the workflow to operate over the Cartesian product of the input collections. This gives rise to the following sets of output values:

- $A = \{f_1(u) \mid u \in U\}$
- $B = \{f_2(a) \mid a \in A\}$
- $C = \{f_3(a, b) \mid a \in A; b \in B\}$

If we examine the pairs of output values $\{(a_i, b_j) \mid a_i \in A; b_j \in B\}$ consumed as inputs by component 3, we see that although the component definition does not express an explicit association between its input channels, an implicit association is introduced by the workflow wiring structure. The only elements from A and B that may be paired are those derived from a common value of u. For example, given $U = \{u_1, u_2\}$:

- $A = \{a_1, a_2\}$, $a_1 = f_1(u_1)$, $a_2 = f_1(u_2)$
- $B = \{b_1, b_2\}$, $b_1 = f_2(f_1(u_1))$, $b_2 = f_2(f_1(u_2))$

In other words, if a user were to manually input the two elements in U, running the workflow each time, (which is the kind of labour that we are trying to automate), Step 3 would execute twice as f_3 (a_1, b_1) and f_3 (a_2, b_2). It would not execute four times using the Cartesian product of all possible pairs of inputs from A and B. We prevent this spurious execution by introducing key-based association.

Key-based association exploits the fact that every data value in a user input collection has a unique and well-defined address, from which the originating component, collection and position within that collection can be deduced. We use the user input addresses to form a key for each result in the system. The key of a user input value is its own address. Any value derived, directly or indirectly, from a user input value, contains the address of that value as part of its key. Thus a key is a list of user input addresses which encodes the provenance of each derived value.

We consider two user input addresses to be comparable if and only if they originate in the same channel of the same component. Two keys k_1 and k_2 are said to be associated if and only if:

– No address a_1 in k_1 is comparable to an address a_2 in k_2, or
– For every address a_1 in k_1 that is comparable to an address a_2 in k_2, $a_1 = a_2$.

That is, two values are associated if and only if they derive from completely distinct lineages, or in the case that they are derived from overlapping sets of channels, they are derived from the same value in each of those channels.

By ensuring that a Processor is only queued for execution if its input values are mutually associated, we preserve the structural integrity of the workflow when it is iterated. We also remove the need for user intervention to specify the association mode.

3.3 Collection Formation: Aggregation and Key-Slicing

GPFlow provides two ways to form a collection: aggregation and key-slicing. An example of aggregation is where we wish to merge the elements of two parallel arrays to form a single array of 2-dimensional vectors. An example of key-slicing is where we wish to gather all values produced by a component to perform some synthesis or summarizing operation.

If an input field has type “Array of T” for some type T, it may be connected to one or more output channels, provided their types are “T” or “Array of T”. When a value is assembled for such an input field, a single sequence is constructed. This sequence contains all constituent elements of the nominated antecedent output channels, subject to the key association criterion described above. This extension alone is sufficient to enable aggregation.

Key-slicing is based on the observation that the collected keys of the result set generated by a component form a discrete hypercube, with dimensionality defined by the set of keys that index the elements of the result set. An output value is associated with each point in this hypercube. If we remove a key field, we project onto a hypercube of lower dimension, each point of which indexes a collection of values, namely those distinguished by the value of the key removed. Intuitively, we take a slice through the result cube.

To implement this in GPFlow, we permit the user to nominate one or more unbound input variables to be removed from the key for a particular input channel. Sliced inputs may belong to the component that contains the output channel or to any of its antecedents. Any sliced input values are selectively ignored when the association test is applied during input value assembly. This provides the user with an intuitive way to specify collections.

4 Case Study: A Phylogenetic Tree Inference Workflow

This paper is focused on the integration of collection processing into scientific workflow systems to enable comparative parametric studies and data synthesis. A good example of the latter is provided by the workflow fragment described below, which mirrors the illustrative case study presented in [3] and provides a clear opportunity for comparison of collection processing as seen by users in GPFlow and Kepler.

The workflow contains the following steps:

1. Get Homologs - user supplies the locus tag of one or more reference genes, and a list of comparison genomes. BLAST [11] is used to identify homologs of the reference genes in the comparison genomes.
2. Get NCR - extracts a DNA sequence adjacent to each of the homologous genes.
3. Align - uses ClustalW [12] to align the DNA sequences produced by step 2, forming a multiple alignment.
4. Phylip Pars - takes the multiple alignment emitted by step 3 and executes the Pars program [13] once for each of a set of distinct seed values. The result set consists of a set of trees.
5. Consensus Tree - uses the key-slice operator to gather the trees inferred by Pars into a collection as required by the Phylip Consense [13] program, which produces a single consensus tree.

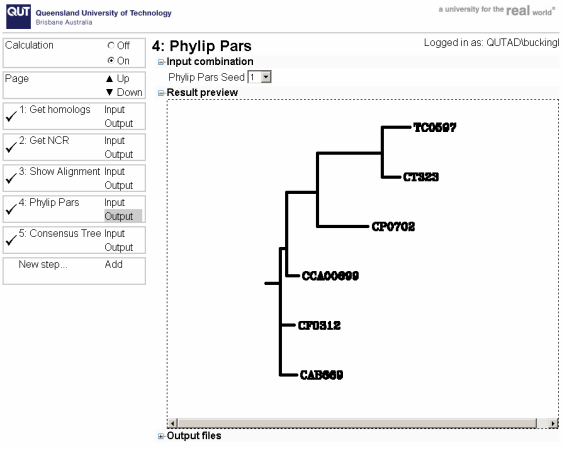

Fig. 2. Consensus tree workflow

Fig. 2 shows one of the trees produced by Phylip Pars resulting from a gene drawn from the Chlamydia trachomatis genome, indexed according to the seed value supplied to Pars. The user may readily view and compare alternate results by means of the drop down menu in the top section of the screen. The navigation component at left allows the user to move between components, adjusting parameters as necessary and triggering restarts of components whose input state has changed.

Although it is not immediately apparent in Fig. 2, this workflow contains an implicit loop in step 4 to generate a forest of distinct trees, which are gathered to form a collection in step 5 by clicking a single check box in the user interface. Use of data driven iteration and key-slicing allows us to hide detail from the scientist, who sees this as a simple linear pipeline of components, each of which has a readily identifiable role in the experiment. The workflow can be described in the language of the scientific domain without recourse to the language of computer science. This stands in contrast with the workflow described in detail in [3], in which the user encounters components such as "TextFileReader", "ExceptionCatcher", "NexusFile-Composer"

and "TextFileWriter", and where explicit control flow primitives potentially introduce further confusion. The actions covered in steps 4 and 5 of the GPFlow workflow require 10 steps in Kepler. Furthermore, a special kind of actor had to be included in the Kepler solution to permit formation of a collection, where this situation is elegantly handled by the use of key-slice aggregation in GPFlow.

The next step for the user, having prototyped this workflow fragment, might be to execute the entire workflow over a range of input genes. This is accomplished by selecting two or more reference genes in the Get Homologs input panel. The workflow result set is then automatically recalculated, with the single consensus tree produced at the end of the run replaced by a corresponding collection of trees. This collection of trees would be available for individual analysis or further aggregated processing. Doing this manually would require the user to laboriously run through the entire workflow for each input gene. To achieve the same result in Kepler would require coding of another explicit loop.

5 Conclusions and Further Work

There is a widely acknowledged need for better eScience tools which enable scientists to concentrate on their research rather than the technology. We have presented a model and system enabling sophisticated comparative studies, vital in eScience, to be devised and undertaken without the need for programming or scripting, or for explicit management of the collection and data associations. A GPFlow workflow defined on a single value may be lifted to operate on a collection of values with no change required to the workflow. The model supports collective operations on aggregated data sets, through workflows encompassing a map reduce style of dataflow, and manages dependencies through a novel mechanism based on key aggregation and slicing. Thus, GPFlow differs from existing approaches in automatically determining the association mode for combinations of collections from the dataflow graph.

As in earlier versions of our system, the workflow is interactive; workflows can be interrupted and changed on the fly, thereby supporting experimentation. Workflows can be published and shared, thus enriching the research environment across a community.

Further work will address integration with Grid computing so that grid services may be easily invoked – currently the system can support arbitrary web services but has no specific facilities for grid computing. A longer term investigation is underway to raise the level of abstraction of the workflow further to enable high level questions and queries to be posed, ultimately including hypothesis generation. We envisage that this will form a layer on top of the current model. Finally we wish to apply the system to other fields, notably the analysis of environmental sensor network data.

The system will be made freely available under a BSD style license. A demonstration system can be accessed from the project home page (`http://www.mquter.qut.edu.au`).

Acknowledgements. We gratefully acknowledge the support of Microsoft Research, the Queensland Government and QUT.

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Sixth Symposium on Operating System Design and Implementation OSDI 2004, USENIX Association, San Francisco, CA (2004)
2. Rygg, A., Roe, P., Wong, O.: GPFlow: An Intuitive Environment for Web Based Scientific Workflow. In: Fifth International Conference on Grid and Cooperative Computing Workshops, pp. 204–211. IEEE Computer Society, Los Alamitos (2006)
3. Fox, G.C., Gannon, D.: Special Issue: workflow in Grid Systems (Editorial). Concurrency and Computation: Practice and Experience 18(10), 1009–1019 (2006)
4. McPhillips, T., Bowers, S.: An Approach for Pipelining Nested Collections in Scientific Workflows. ACM SIGMOD Record, ACM SIGMOD/PODS 34(3), 12–17 (2005)
5. McPhillips, T., Bowers, S., Ludäscher, B.: Collection-Oriented Scientific Workflows for Integrating and Analyzing Biological Data. In: Leser, U., Naumann, F., Eckman, B. (eds.) DILS 2006. LNCS (LNBI), vol. 4075, pp. 248–263. Springer, Heidelberg (2006)
6. Oinn, T., Greenwood, M., Addis, M., Alpdemir, M.N., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M., Senger, M., Stevens, R., Wipat, A., Wroe, C.: Taverna: lessons in creating a workflow environment for the life sciences. In: Concurrency and Computation: Practice and Experience, vol. 18, pp. 1067–1100. Wiley InterScience, Chichester (2006)
7. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Computing Surveys 36(1), 1–34 (2004)
8. Davis, A.L., Keller, R.M.: Data flow program graphs. IEEE Computer 15(2), 26–41 (1982)
9. Hylands, C., Lee, E., Liu, J., Liu, X., Neuendorffer, S., Xiong, Y., Zhao, Y., Zheng, H.: Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M02/25, University of California, Berkeley (2003)
10. Stevens, R., Robinson, A., Goble, C.A.: myGrid: Personalized Bioinformatics on the Information Grid. Bioinformatics, Oxford Journals 19(suppl. 1), i302-i304 (2003)
11. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. J. Mol. Biol. 215(3), 403–410 (1990)
12. Thompson, J.D., Higgins, D.G., Gibson, T.J.: CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res. 22, 4673–4680 (1994)
13. Felsenstein, J.: PHYLIP (Phylogeny Inference Package) version 3.6. Distributed by the author. Department of Genome Sciences, University of Washington, Seattle