Parallel Computing Patterns for Grid Workflows

Cesare Pautasso, Gustavo Alonso
Department of Computer Science
ETH Zurich
ETH Zentrum, 8092 Zürich, Switzerland
{pautasso,alonso}@inf.ethz.ch

Abstract

Whereas a consensus has been reached on defining the set of workflow patterns for business process modeling languages, no such patterns exists for workflows applied to scientific computing on the Grid. By looking at different kinds of parallelism, in this paper we identify a set of workflow patterns related to parallel and pipelined execution. The paper presents how these patterns can be represented in different Grid workflow languages and discusses their implications for the design of the underlying workflow management and execution infrastructure. A preliminary classification of these patterns is introduced by surveying how they are supported by several existing advanced scientific and Grid workflow languages.

1 Introduction

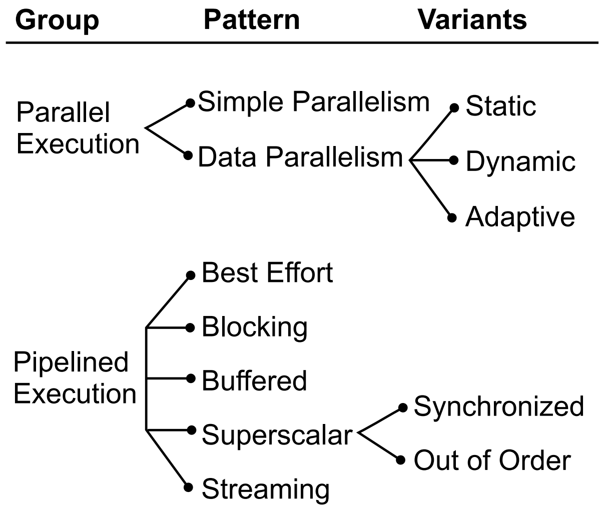
Scientific and Grid workflow languages [16, 23, 42] make use of techiques such as massively parallel execution and pipeline processing [18] to provide scientists with powerful modeling primitives and language constructs. These primitives are used to implement parallel task execution while retaining the characteristic high abstraction level of workflow languages.

For example, the notion of data flow used in scientific workflows is a natural representation for simple data processing pipelines. It has the advantage that parallel execution of independent tasks is modeled for free [19]. Pure data flow, however, is not expressive enough to model either branches and merges in the execution path nor iterative behavior [26]. This is why workflow languages typically focus on the control flow primitives rather than on the data flow aspects. An example of this focus is the existing literature on control flow patterns [37]. In scientific applications, however, data flow [34] and parallel computing patterns play a crucial role, not only in terms of design-time modeling but also in important performance optimization aspects related to run-time execution at a large-scale.

In this paper, as a first step to better understand the relationship between parallel computing and scientific workflows, we define a set of language patterns. These patterns can be classified in two broad categories: Parallel Execution and Pipelined Execution (Table 1). Parallel execution patterns include: 1) *Simple parallelism*, where tasks lacking control flow dependencies are executed in parallel; 2) *Data parallelism*, a form of single instruction multiple data (SIMD) parallelism [15] with three variants: static, dynamic and adaptive. Pipelined execution patterns include: 3) *best effort* pipelines, where intermediate results are dropped if downstream tasks are not ready to process them; 4) *blocking* pipelines, where a form of flow control is used to stop tasks that are located upstream from busy ones; 5) *buffered* pipelines, where the workflow accumulates intermediate results between tasks; 6) *superscalar* pipelines, where multiple parallel instances of slow tasks are started to process intermediate results; 7) *streaming* pipelines where intermediate results are fed into continuously running tasks.

The purpose of this paper is not just to introduce these patterns, as most of them have already been featured in parallel computing languages for a long time (e.g., HeNCE [5]). The goal instead is to classify them in such a way that their practical implications for Grid workflows rather than for parallel programming languages become more clear. A first use of these patterns could be to study and compare the expressive power of existing scientific and Grid workflow languages. Yet, a more interesting application will be to revisit different aspects of the management of scientific workflows with these parallel computing patterns in mind.

Thus, supporting these parallel computing patterns in a workflow system involves dealing with issues related to their efficient implementation. Parallel and pipelined ex-



Group	Pattern	Variants
Parallel Execution	Simple Parallelism	
	Data Parallelism	Static
		Dynamic
		Adaptive
Pipelined Execution	Best Effort	
	Blocking	
	Buffered	
	Superscalar	Synchronized
		Out of Order
	Streaming	

Table 1. Overview of the parallel and pipelined execution scientific workflow patterns

ecution patterns are used to introduce performance optimizations in the workflow [25]. Still, these optimizations need to be integrated with the rest of the features (e.g., persistent execution, data management, task scheduling, data provenance, and lineage tracking) provided by a scientific workflow system. More concretely, partitioning a dataset a thousand-fold may indeed provide a comparable speedup in the execution of the workflow. However, the overhead of collecting and managing the corresponding amount of metadata should not be underestimated. By making these parallel computing patterns explicit, we hope to facilitate the discussion on what needs to be done to implement them efficiently.

The rest of the paper is structured as follows. For each group of patterns (Parallel Execution, Section 2, and Pipelined Execution, Section 3), we include a general description of its purpose, we present some alternatives on how it can be modeled (depending on the characteristics of the scientific workflow language) and sketch some possible extensions and variations. Given the large number of different scientific and Grid workflow languages that have been proposed in the literature [16, 23, 42], we also discuss to which extent a set of representative ones supports the patterns introduced in the paper. Section 4 discusses some related work and Section 5 concludes the paper.

2 Parallel Execution

2.1 Simple Parallelism

Parallel execution is the simplest technique to reduce the execution time of a scientific workflow. Given a set of tasks, if there are no dependencies between them and enough Grid computing resources are available, they can be scheduled for execution using multiple parallel threads of control.

Most languages based on data flow (e.g., SCIRun [30] and KEPLER [3]) support this pattern in a straightforward manner (Figure 1a). The data flow graph of a process explicitly defines what data is exchanged between tasks. It can be analyzed to infer the partial order of execution of the tasks as follows. If a pair of tasks exchange data, they depend on each other and must be executed sequentially, i.e., the task producing data as output is followed by the task requiring it as input. For example, a task writes its results into a file and the following one can read from the file only after it has been closed. Likewise, a task can invoke a Web service only after the response from the previous Web service has been received. If a pair of tasks is not linked by such data dependency, then both tasks can be scheduled for concurrent execution.

Simple parallel execution can also be expressed using the control flow. Languages modeling control flow with a simple directed graph (e.g., YAWL [36], or JOpera [31]) still represent this pattern in an implicit way, where the lack of control flow dependencies (Figure 1b) between tasks implies the lack of ordering constraints in the execution. This is similar to the semantics of some languages (e.g., GEL [21]), where simple parallelism is intended as a synonym of *lack of dependencies* between tasks. As a consequence, the workflow engine has the possibility (but it is not compelled) to run independent tasks in parallel. Thus, the actual concern of implementing parallel execution is shifted to the scheduling system, which can adapt the degree of actual parallelism based on the available execution resources.

In other cases, simple parallelism is modeled using adhoc control flow constructs, e.g., the *parallel split* node in UML Activity Diagrams (Figure 1c), where the flow is still represented as a directed graph. An example of an hybrid approach (where the nodes of the control flow graph can be nested into blocks) can be found in [40], where various alternative representations of parallelism are analyzed for the standard Business Process Modeling Notation (BPMN [28]). In this case the tasks contained inside a *parallel box* are executed in parallel, only after the predecessor of the parallel box has completed (Figure 1d). A final alternative to model the simple parallelism pattern is taken by languages following a pure block-based ap-

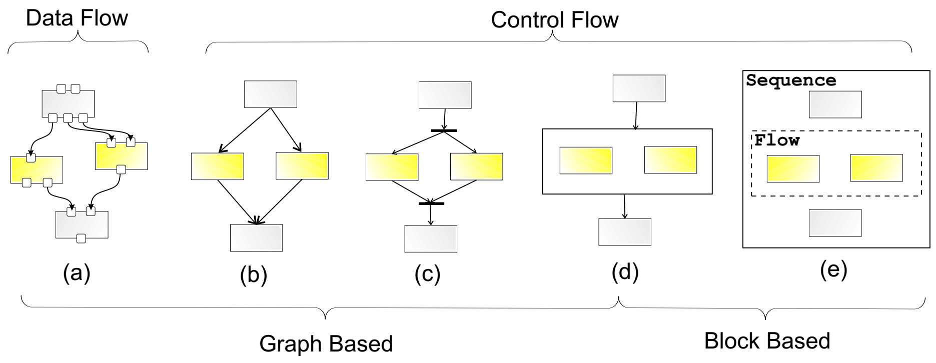


Figure 1. Alternative representations of the simple parallelism pattern

proach (Figure 1e), e.g., like `flow` structured activities *à la* XLANG/BPEL [27] or `parallel` blocks like in Karajan [39].

From these examples it can be seen that, in some cases, modeling parallelism with the control flow requires the modeler to make an additional effort as parallelism must be expressed explicitly. Compared to data flow, using the control flow gives more control over the actual order of task execution. The advantage is that sometimes it may be required to actually reduce the amount of parallelism in the workflow by modeling dependencies between tasks which are not visible in the data flow of a process. This is not possible to express in pure data flow languages, unless an additional representation of the control flow is included. This can be overlayed, like in TAVERNA [41], or shown side-by-side, like in JOpera [31].

2.2 Data Parallelism

Single Program Multiple Data streams (SPMD [18]) is a parallel processing technique where the same program (or task, in workflow terminology) is applied to multiple data elements. All elements are processed in parallel, if – as before – no dependencies exist among them.

The *Data Parallelism* pattern applies this idea to speed up the execution of workflows that are run over large input datasets. Instead of feeding the entire dataset to a task, the data is partitioned and a copy of the same task is applied in parallel over each (independent) partition. Once all data partitions have been scheduled for execution, there can be different synchronization semantics to proceed with the execution (wait for all, wait for one, n-out-of-m).

The pattern is often used to model embarrassingly parallel computations such as parameter-sweep simulations [1]. This pattern has also a close relationship to the classical *multi-instance* workflow pattern [37] and its variations.

Before presenting several examples on how this pattern can be modeled, we further qualify Data Parallelism by defining when the degree of parallelism is determined (at design-time, or at run-time) and by whether it can be controlled from within the workflow itself (manual vs. adaptive).

2.2.1 Static vs. Dynamic Data Parallelism

If the number of partitions is known in advance (i.e., at design-time) and it is fixed for all workflow executions, any workflow language that supports the simple parallelism pattern can be used to model the static data parallelism pattern [37].

The pattern becomes more challenging to model if the number of partitions can only be determined at run-time. This is an important aspect to ensure the portability of a workflow definition across multiple execution environments of different sizes [13]. A workflow should not be designed to run on a specific Grid execution environment. Instead, key environment properties should be abstracted in a set of workflow parameters bound as late as possible, i.e., at deployment or at run-time. The degree of partitioning of an input dataset for a task is one of such parameters. Its value depends both on the actual size of the dataset and on the amount of computing resources that are available when the workflow is executed.

2.2.2 Adaptive Data Parallelism

In the simplest case, the number of partitions can be manually controlled by passing it as input to the workflow. Alternatively it can be automatically estimated during the execution of the workflow itself.

As a motivation for this pattern, we present the model shown in Figure 2. This model indicates that the total execution time of a workflow that uses data parallelism to speed

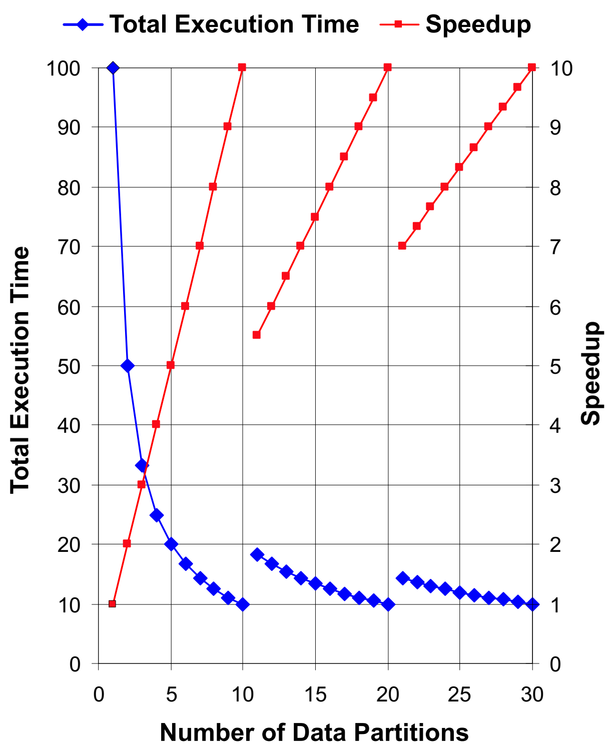


Figure 2. Adaptive Data Parallelism: execution time and speedup with a variable number of data partitions of the same dataset scheduled on a fixed amount of 10 homogeneous execution resources

up the processing of an input dataset of fixed size, is highly sensitive to the number of data partitions (d, x-axis) with respect to the amount of available resources ($R = 10$). Assuming a homogeneous set of partitions, for $d \leq R$ the speedup increases linearly with d. For $d = R+1$, however, the speedup drops almost by 50%. In general, this effect is most prominent when $d = kR + 1$ ($k > 0$). Given that – in this case – one partition cannot be executed concurrently with the others, the performance of the workflow suffers due to the unbalanced use of the available resources. Since the model assumes that the overhead of scheduling each partition is negligible, the maximum speedup is reached again for $d = kR$ ($k > 1$). For this case, the execution time does not change with a larger number of partitions because the size of each partition decreases proportionally.

Considering this staircase effect, in order to ensure the balanced execution of the workflow, a Grid workflow system can provide support for the adaptive data parallelism pattern. With it, the workflow is dynamically adapted to the current state of the Grid execution environment [32]. The optimal number of partitions is determined automatically as a function of the number of available Grid resources. As an example, this estimate can be based on the following heuristic: the number of partitions d should be a multiple of the number of available processors R, so that full resource utilization is ensured by scheduling each partition for parallel execution.

Modeling such *adaptive data partitioning* strategy requires a workflow language to support some form of reflection. Through reflection, the workflow reads information about the environment to automatically steer its execution (in the dynamic case) or adapt its structure (in the static case). This can be further extended by using resource reservation capabilities provided by advanced resource management and scheduling systems [10]. By doing this, not only the optimal data partitioning is determined for a task but the workflow also ensures that enough resources are reserved to process the resulting set of tasks.

A further distinction can be made by observing that not all data partitions need to be of the same size. We can distinguish between a *homogeneous* data partitioning strategy and a *heterogenenous* data partitioning strategy. In an heterogenenous execution environment, splitting the data unevenly may be required to ensure that all tasks run to completion in the same amount of time [2].

The downside of adaptive data parallelism is that the runtime structure of a workflow instance is not only influenced by its input data, but also by the current properties of the Grid execution environment. This has dear implications for the workflow engine in terms of maintaining lineage metadata and guaranteeing deterministic memoization [22].

2.2.3 Modeling Examples for Data Parallelism

The main challenge of modeling the data parallelism pattern consists of dealing with a variable number of tasks, whose number may change for each workflow execution. Different languages follow different approaches (Figure 3): 1) Static and Dynamic Graph Rewriting; 2) First-order functions; 3) Multiple job submissions per task; 4) Repeated asynchronous job submissions; 5) Parallel for-each blocks.

Graph rewriting is a technique based on the replacement of a task node with n copies of itself and the necessary rewiring of the corresponding control and data flow. If this is applied at design (or deployment) time, like in the case of Triana [35], this technique is similar to a form of macro expansion and can only be applied to support the static variant of this pattern.

In the dynamic case, a vector is passed as input to a task. At run-time, for each element of the vector, a new instance of the task will be created and started in parallel. Since the

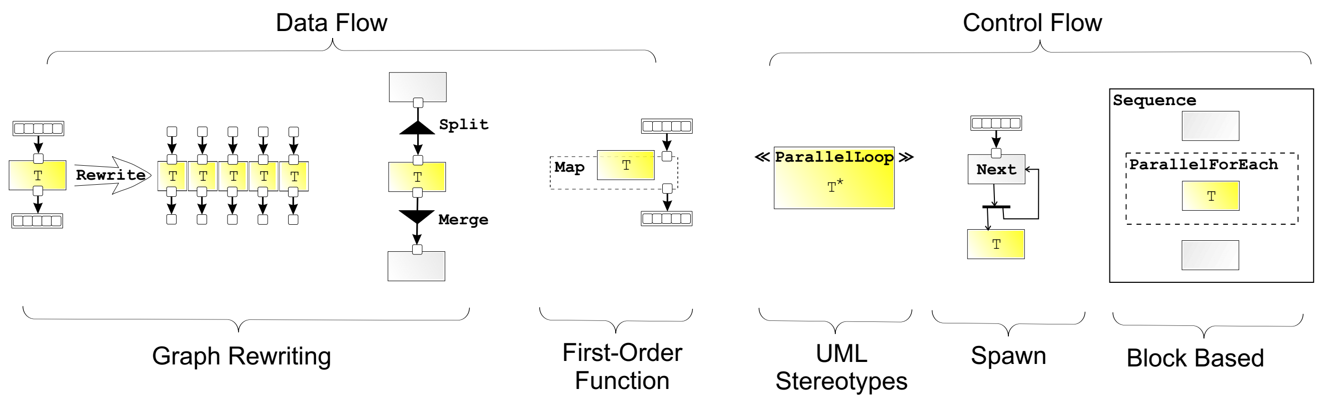


Figure 3. Modeling Examples for the Data Parallelism Pattern

number of elements in the vector is known only at run-time, the number of parallel instances of the task may change for every execution of the workflow. In case of a task receiving multiple input vectors, their elements can be joined for each task instance. This can be done assuming that each vector has the same number of elements, like, for example, in JOpera [31]. Alternatively, the latest version of Taverna can also iterate over the cartesian product of the vectors [41], which can then have different lengths.

An example of dynamic parallel loops, which however only models concurrency from the control flow perspective is presented in Teuta [33]. In this case, UML activities are stereotyped with the "parallelloop" tag. Furthermore, an asterisk ("*") is used to denote the dynamic instantiation of a variable number of activities. Still, no explicit means of controlling the number of partitions is presented.

From a syntactical point of view, this approach is similar to the following one, which does not require to modify the structure of the workflow to deal with a variable number of partitions. The idea consists of relaxing the assumption that each task models the scheduling and the execution of a single job. If a task can be used to model the batch submission of a set of jobs to a scheduler, then the workflow does not need to grow because only one instance of a task is enough to control the execution of multiple parallel jobs. Synchronization and retrieval of the results is also simplified, as the same batch submission facilities of the scheduler can be used to retrieve the results of all completed jobs. As an example, JOpera can perform multiple job submissions to a scheduler within the same task. Experiments have shown that given a non-neglibible job submission overhead it pays off to submit a large number of parallel jobs as a batch within the same task [11].

Languages (e.g., KEPLER [3]) based on data flow can also use an approach based on functional programming, where a first-order function (in this case, the $list' = Map(task, list)$ function) is used to apply in parallel the same $task$ (or mapping) over each element of a $list$ passed as input to the first-order Map function [12].

Languages supporting control flow loops and having the ability to *spawn* tasks (i.e., asynchronously submit a task to a scheduler without waiting for its completion) can combine these two constructs to iterate over an input vector and spawn a new task over each of its elements. With this solution, however, given the lack of a specific language construct, synchronization over all submitted tasks becomes difficult to express [17]. Since all tasks are executed asynchronously with respect to the workflow their results cannot be easily accessed in order to further process them in the rest of the workflow.

Block based languages (e.g., the Abstract Grid Workflow Language (AGWL) [14]) use a special *parallel for* or *parallel for-each* block to specify that the tasks contained within the block can be iterated over in parallel. This is also the case for Karajan [39], where a parallel *mode* for the `for` loop construct is included. A parallel option for the `forEach` structured activity is also included in the latest version of the WS-BPEL 2.0 [27] specification where different synchronization strategies can be modeled by customizing the completion condition associated with the `forEach` activity. Similar to the other approaches, the body of the loop must not depend on results of previous iterations.

A compact notation for modeling the parallel-for block construct has been described for the Grid Execution Language (GEL [21]) as follows: $\sum_{x=1}^{d} P(x)$ indicating that task P is to be executed d times in parallel. GEL provides explicit support for both static and dynamic data parallelism. While the number of iterations of a `pfor` must be known at compile-time (static) the number of iterations of a `pforeach` can depend on the results of previous tasks or

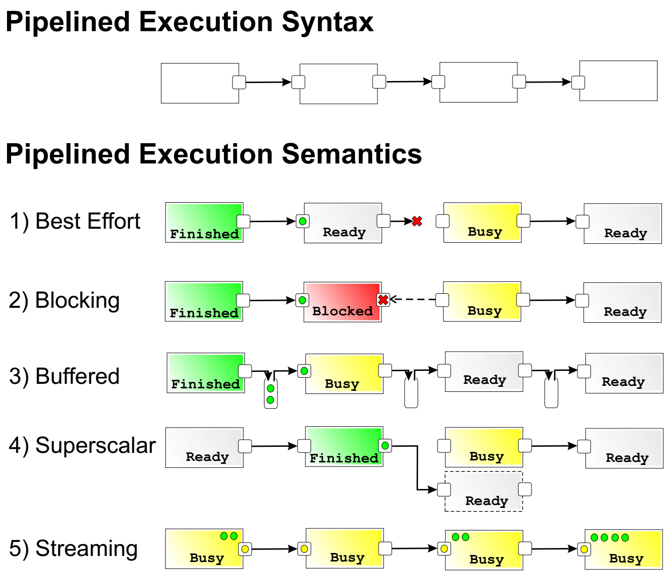


Figure 4. Informal representation of different pipelined execution patterns

on the state of the execution environment (e.g., the number of files listed in a folder), thus also supporting the dynamic data parallelism pattern.

3 Pipelined Execution

Whereas using the Data Parallelism pattern can speed up the execution of one or more tasks applied *in parallel* to a vector of input data elements, the Pipelined Execution pattern comes into play when a sequence of more than one tasks is applied *sequentially* to a vector of input data elements.

As opposed to iterative execution, where each data element would have to go through the entire sequence of tasks before the next data element of the vector is processed, with pipelined execution the elements are streamed through the workflow. Similar to the previous pattern, the set of input elements may be known in advance (either at design-time or before the execution of the pipelined sequence begins) or new elements may appear while the pipeline is running (e.g., if a scientific workflow is used to process measurements produced in real-time by a sensor or another source of streaming data).

Thanks to such overlapped parallel execution, applying this pattern can reduce the overall execution time of the workflow by a factor proportional to the length of the sequence of pipelined tasks. This result holds assuming a homogeneous set of tasks and data elements, as well as the availability of a dedicated computing resource for the repeated execution of each task.

From these assumptions stem most of the difficulties in implementing pipelined execution for a workflow language. Given the heterogeneity of the tasks involved and the variability of the input data elements, it cannot be assumed that all tasks of the pipeline take exactly the same time to process their input. Thus, pipeline collisions (when a task sends data to another task that is busy) can be dealt with at the level of the workflow, or within its tasks, or by the application itself. As shown in Figure 4, the workflow can either block the execution of tasks, manage buffers between tasks, or start multiple task instances in parallel. The tasks can assume the role of continously running operators over a data stream. The application can – in some cases – tolerate loss of data.

3.1 Best Effort Pipelines

In the simplest case, no guarantees are provided and data is simply dropped in case of pipeline collisions. This best-effort solution may be satisfactory where it is not required that all input elements are fully processed by the pipeline. For example, real-time constraints on the freshness of the results to be delivered by the workflow may dictate that older data elements are dropped if a task is lagging behind.

Best effort pipelines are a pragmatic approach that greatly simplifies the workflow engine implementation. However, they require careful handling as the execution is no longer deterministic and reproduceable.

3.2 Blocking Pipelines

A first solution to avoid loss of data is to block the execution of a task of the pipeline if its successor is busy. More precisely, this involves an inversion of the control flow between tasks. With blocking semantics, a task can only be (re)started if the predecessor has completed *and* if its successor is idle. This additional constraint ensures that the data produced by a task will not be overwritten, as the downstream tasks will be ready to process it. One example of a system implementing the blocking pipelined execution pattern are KEPLER and JOpera.

Although blocking already ensures that no data is lost within the pipeline, slow tasks will recursively block their predecessors quickly becoming the bottleneck of the entire execution. Another problem concerns the data that is streamed into the pipeline. This can be either pulled into the workflow synchronously with the pipeline execution, or pushed into the workflow at a fixed sampling rate, independent of whether the first task of the pipeline is blocked or

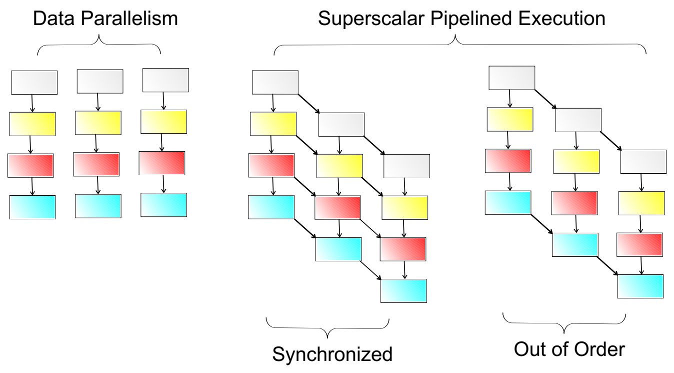


Figure 5. Comparison of the Data Parallelism with the Superscalar Pipelined Execution patterns (Control flow dependencies shown as edges)

ready to receive it. Only in the latter case, data loss may still occur as the source of streaming data cannot be blocked.

Both of these issues can be addressed by generalizing this pattern to use a buffer for storing both the input data and the intermediate results of the pipeline.

3.3 Buffered Pipelines

Providing buffering semantics requires to store and accumulate all intermediate results produced by a task in case the one following it in the pipeline is not yet ready to process a new input data element. Examples of languages that support the buffered pipelines pattern are SCIRun [30], Triana [35], and KEPLER [3].

There are two main challenges that concern the efficient implementation of buffering. Although, from a conceptual point of view, a buffer may effectively decouple a fast producer from a slow consumer, a concrete buffer has a limited (or finite) capacity. Thus, buffering still requires the engine to block upstream tasks in case the buffer gets full. From this, it can be seen that this semantics is a good solution for decoupling tasks whose execution time varies depending on the particular data element of the stream.

The second challenge concerns the interference of a data flow related aspect (i.e., the buffer) and the control flow that defines when a task of the pipeline is ready to be executed. More specifically, in order to start the execution of a task, the engine should not just look whether its control flow dependencies are satisfied but also take into account the state of its data flow buffers.

3.4 SuperScalar Pipelines

Superscalar execution semantics requires to dynamically create additional task instances whenever they are needed to avoid collisions. Thus, if a new input data element is available for a busy task, another instance of the same task is created in order to process the new data element in parallel. This is the approach followed by HeNCE [5].

Creating additional task instances opens up the possibilty of data elements to overtake each other, as some may be delayed in the pipeline. Thus, within the pipeline, for each element synchronization can be enforced between all tasks (full synchronization), or only at the beginning and at the end of the pipeline (out of order), thus leaving the engine the possibility to change the order in which the various elements are processed by intermediate tasks.

More in detail, in Figure 5 we compare these two alternatives with the basic data parallelism pattern (left). With the synchronized variant (center), dependencies are inserted both along the direction of the pipeline, but also between all task instances responsible for processing each data element. This ensures that all elements are processed in strict sequential order. In the other case (out of order, Figure 5 right), dependencies are only inserted where they are needed (i.e., at least at the beginning and at the end of the pipeline). This allows some degree of out-of-order execution, where elements can overtake one another within the pipeline while still ensuring the ordering of the final results.

This technique adds considerable complexity to the underlying engine. In addition, it requires very efficient mechanisms if the progress of the execution is to be made persistent or to actually display it in a monitoring tool.

3.5 Streaming Pipelines

Adding streaming semantics to a pipeline of tasks requires to relax the basic assumptions of having tasks that support a simple request-response interaction, where a task reads its input as it starts and produces output once it is finished. Given the goal of passing input data to a task as soon as it is produced by its predecessor (and avoid the blocking semantics), the workflow cannot wait for the task to finish before restarting it with the next element (like with the buffering semantics) or start another parallel instance (superscalar semantics). Thus, only if tasks allow a more flexible interaction based on multiple requests and multiple responses, the streaming semantics can be fully supported by a workflow language.

An example of the pipelined workflow pattern with streaming semantics can be found in the OSIRIS/SE [6], KEPLER [3] and PTOLEMY [20] systems, as well as in the

Language	Simple Parallelism	Data Parallelism	Pipelining Semantics
Visual Parallel Computing Languages			
HeNCE [5]	Control Flow Graph	Dynamic	Superscalar
SCIRun [30]	Data Flow Graph	Static	Buffered
Scientific and Grid Workflow Languages			
AGWL [14]	Control Flow	Dynamic	
GEL [21]	Control Flow	Dynamic	
JOpera [31]	Data and Control Flow Graph	Adaptive	Best Effort and Blocking
Karajan [39]	Control Flow	Dynamic	
KEPLER [3]	Data Flow Graph	Dynamic	Streaming, Blocking and Buffered
TAVERNA [41]	Data and Control Flow Graph	Dynamic	
Teuta (UML 1.1) [33]	Control Flow	Dynamic	
Triana [35]	Data Flow	Static	Buffered
Business Process Modeling Languages			
BPEL4WS 1.1	Control Flow Block	Static	
WS-BPEL 2.0	Control Flow Block	Dynamic	
BPMN [28]	Control Flow	Dynamic	
Osiris/SE [7]	Control Flow	Static	Streaming (with Buffering)
UML 2.0 [29]	Control Flow Graph	Static	Streaming
YAWL [36]	Control Flow Graph	Dynamic	

Table 2. Summary of the patterns supported by the languages surveyed in this paper

latest version (2.0) of UML activity diagrams [29]. More specifically, in [24] the KEPLER system is extended to support pipelined execution over nested data collections. To do so, a new type of *actor* (or task) is introduced. In order to process a collection of stream elements, the *CollectionActor* interacts with the workflow engine using a special push mechanism based on callbacks. In UML 2.0, the input and output parameters of activities can be flagged as streaming. This way, during the same execution of the activity multiple input and output tokens can be exchanged with the rest of the workflow over these streaming parameters.

As it can be seen from these examples, streaming pipelines break with the black box approach of workflow languages since the properties of the workflow are determined by the tasks themselves. Such deep integration of task semantics into the workflow language also affects the design of the underlying workflow engine, since data must be exchanged with tasks as they are running [7].

4 Related Work

In the context of business process modeling languages there exists a large body of literature on workflow patterns. In [37] the authors focus on control flow aspects, and identify several patterns (e.g., parallel split and multiple instances) which – as we showed in Section 2 – are highly relevant for scientific and Grid workflows. More recently, data [34] and messaging [4] patterns have also been studied extensively. Furthermore, several contributions discuss how specific workflow languages support the various patterns that have been identified (e.g., see [38] as a starting point).

Relatively less work can be found regarding patterns for scientific and Grid workflows, although the need for such systematic classification has already been recognized for some time [8, 9].

In [42] a large number of Grid workflow systems are surveyed, evaluated and classified according to several dimensions, including the support for basic workflow patterns (i.e., the presence of sequential and parallel control structures, DAGs vs. loops, etc.). However, the more advanced patterns presented in this paper concerning data parallelism and pipelined execution are not covered.

A preliminary classification of workflow patterns related to parallel computing can be found in [25]. Inspired by database query processing systems, the authors identify three different kinds of parallelism: *inter-workflow*, *intra-workflow* and *intra-program*. Intra-program and inter-workflow parallelism are also mentioned in [21] under the terms of *fine-grained* and *coarse-grained* parallelism. These can be compared to the patterns described in this paper as follows.

Inter-workflow parallelism refers to the simultaneous execution of multiple workflow instances (in general) and of multiple instances of the same workflow (in particular). With this definition, inter-workflow parallelism can be seen as a feature shared by most scientific and Grid workflow management systems which does not affect the expressive power of the corresponding workflow modeling language.

Intra-workflow parallelism is defined as the concurrent execution of more than one program (or task) within the same workflow. In this paper we further distinguish tasks without dependencies (parallel execution patterns) from tasks depending on the previous results of one another (pipelined execution patterns).

Intra-program parallelism implies the distributed execution of individual tasks of the workflow. Due to its fine granularity, intra-program parallelism falls below the modeling capabilities and scope of a workflow language. Instead, it requires special support from the workflow system to correctly stage the distributed execution of the task.

5 Conclusion

In this paper we discuss several parallel computing patterns that are crucial for optimizing the performance of large scale scientific and Grid workflows. For each pattern, we motivate its purpose, present some modeling alternatives (depending on the characteristics of the workflow language) and sketch some possible extensions and variations. Currently, as shown in Table 2, no existing language and workflow execution engine provide support for all variations of the patterns we have identified. Still, all languages support simple parallelism, and thus static data parallelism. From our survey, we also observed that dynamic data parallelism and pipelined execution can be considered as orthogonal language features, as there are languages which support either one (e.g., Triana or Osiris/SE), or both (e.g., Kepler, JOpera, and HeNCE). In order to highlight the main challenges of providing an efficient implementation, we have also commented on the patterns' implications on the design of the underlying workflow engines.

Acknowledgements

The authors would like to thank Biörn Biörnstad and the anonymous reviewers for their valuable feedback. Part of this work is funded by the European IST-FP6-004559 project SODIUM (Service Oriented Development In a Unified fraMework) and the European IST-FP6-15964 project AEOLUS (Algorithmic Principles for Building Efficient Overlay Computers).

References

[1] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod/G: killer application for the global grid? In *Proc. of the 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 520–528, Cancun, Mexico, 2000.

[2] G. Alonso, W. Bausch, C. Pautasso, M. Hallett, and A. Kahn. Dependable Computing in Virtual Laboratories. In *Proc. of the 17th International Conference on Data Engineering (ICDE2001)*, pages 235–242, Heidelberg, Germany, 2001.

[3] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludaescher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientifi c Workfbws. In *Proc. of the 16th Intl. Conference on Scientifi c and Statistical Database Management (SSDBM)*, Santorini Island, Greece, June 2004.

[4] A. Barros, M. Dumas, and A. H. ter Hofstede. Service Interaction Patterns. In *Proceedings of the 3rd International Conference on Business Process Management (BPM2003)*, volume 3649 of *LNCS*, pages 302–318. Springer, September 2005.

[5] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing. In *Proc. of the 1991 ACM/IEEE conference on Supercomputing*, pages 435–444, Albuquerque, New Mexico, 1991.

[6] G. Brettlecker, H. Schuldt, and R. Schatz. Hyperdatabases for Peer-to-Peer Data Stream Processing. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, pages 358–366, San Diego, USA, 2004.

[7] G. Brettlecker, H. Schuldt, and H. Schek. Towards Reliable Data Stream Processing with OSIRIS-SE. In *Proceedings of the Databases for Business, Techology and Web Conference (BTW 2005)*, pages 405–414, Karlsruhe, Germany, 2005.

[8] J. Chen and Y. Yang. Key Research Issues in Grid Workfbw Verifi cation and Validation. In *Proceedings of the Fourth Australasian Symposium on Grid Computing and e-Research (AusGrid 2006)*, pages 97–104, Hobart, Australia, January 2006.

[9] D. Cybok. A Grid workfbw infrastructure. *Concurrency and Computation: Practice and Experience*, December 2005. DOI: 10.1002/cpe.998.

[10] K. Czajkowski, I. Foster, and C. Kesselman. Resource Co-Allocation in Computational Grids. In *Proc. of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 219–228, 1999.

[11] O. Deak. Grid Service Execution for JOpera. Master Thesis, Department of Computer Science, ETH Zurich, 2005.

[12] J. Dean and S. Ghemawat. MapReduce: Simplifi ed Data Processing on Large Clusters. In *Proc. of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 137–150, San Francisco, USA, 2004.

[13] E. Deelman *et al.* Pegasus: Mapping Scientific Workflows onto the Grid . In *Proc. of the 2nd European Across Grids Conference*, pages 11–20, Nicosia, Cyprus, 2004.

[14] T. Fahringer, J. Qin, and S. Hainzer. Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In *Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGrid2005)*, Cardiff, UK, May 2005.

[15] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. on Computers*, C-21(9):948–960, September 1972.

[16] D. Gannon, G. Fox, A. Farazdel, C. Goble, E. Deelman, and D. Berry, editors. *Workflow in Grid Systems Workshop at the Global Grid Forum (GGF10)*, Berlin, Germany, March 2004.

[17] T. Glatard, J. Montagnat, and X. Pennec. Grid-enabled workflows for data intensive medical applications. In *Proceedings of the 18th IEEE International Symposium on Computer-Based Medical Systems (CBMS'05)*, pages 537–542, June 2005.

[18] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

[19] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36(1):1–34, March 2004.

[20] E. A. Lee and T. M. Parks. Dataflow Process Networks. In *Proceedings of the IEEE*, volume 83, pages 773–801, May 1995.

[21] C. C. Lian, F. Tang, P. Issac, and A. Krishnan. GEL: Grid Execution Language. *Journal of Parallel and Distributed Computing* ., 65(7):857–869, July 2005.

[22] D. T. Liu and M. J. Franklin. The Design of GridDB: A Data-Centric Overlay for the Scientific Grid. In *Proc. of the Thirtieth International Conference on Very Large Data Bases*, pages 600–611, Toronto, Canada, 2004.

[23] B. Ludäscher and C. Goble, editors. *Special Section on Scientific Workflows*, volume 34 of *SIGMOD RECORD*, September 2005.

[24] T. M. McPhillips and S. Bowers. An approach for pipelining nested collections in scientific workflows. *SIGMOD Record*, 34(3):12–17, September 2005.

[25] L. A. Meyer, S. C. Rössle, P. M. Bisch, and M. Mattoso. Parallelism in Bioinformatics Workflows. In *Proceedings of the 6th International Conference on High Performance Computing for Computational Science (VECPAR2004)*, volume 3402 of *LNCS*, pages 583–597. Springer, June 2004.

[26] M. Mosconi and M. Porta. Iteration constructs in data-flow visual programming languages. *Computer Languages*, 26(2–4):67–104, July 2000.

[27] OASIS. *Web Services Business Process Execution Language (WSBPEL) 2.0*, 2006.

[28] OMG. *BPMN: Business Process Modeling Notation 1.0*. Object Management Group, 2004. `http://www.bpmn.org`.

[29] OMG. *Unified Modeling Language (UML) 2.0*. Object Management Group, August 2005. `http://www.uml.org`.

[30] S. G. Parker and C. R. Johnson. SCIRun: A Scientific Programming Environment for Computational Steering. In *Proc. of the 1995 ACM/IEEE Supercomputing Conference*, 1995.

[31] C. Pautasso and G. Alonso. The JOpera Visual Composition Language. *Journal of Visual Languages and Computing*, 16(1–2):119–152, 2004.

[32] B. Plale, D. Gannon, Y. Huang, G. Kandaswamy, S. L. Pallickara, and A. Slominski. Cooperating Services for Data-Driven Computational Experimentation. *CiSE, Computing in Science and Engineering*, 7(5):34–43, September 2005.

[33] S. Pllana, T. Fahringer, J. Testori, S. Benkner, and I. Brandic. Towards an UML Based Graphical Representation of Grid Workflow Applications. In *Proc. of the 2nd European Across Grids Conference*, pages 149–158, Nicosia, Cyprus, 2004.

[34] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In *Proc. of the 24th International Conference on Conceptual Modeling (ER 2005)*, pages 353–368, Klagenfurt, Austria, 2005.

[35] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana Applications within Grid Computing and Peer to Peer Environments. *Journal of Grid Computing*, pages 199–217, June 2003.

[36] W. van der Aalst and A. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[37] W. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.

[38] W. M. P. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–85, 2003.

[39] G. von Laszewski and M. Hategan. Workflow Concepts of the Java CoG Kit. *Journal of Grid Computing*, January 2006.

[40] S. A. White. *Process Modeling Notations and Workflow Patterns (Workflow Patterns with BPMN and UML)*. OMG Whitepaper, January 2004.

[41] K. Wolstencroft *et al.* Panoply of Utilities in Taverna. In *Proc. of the 1st IEEE International Conference on e-Science and Grid Computing*, pages 156–162, 2005.

[42] J. Yu and R. Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 2006 (to appear).