Workflow Fine-grained Concurrency with Automatic Continuation

Giancarlo Tretola[1] and Eugenio Zimeo[2]

[1]University of Sannio
Department of Engineering
Benevento, 82100 ITALY
tretola@unisannio.it

[2]University of Sannio
Research Centre on Software Technology
Benevento, 82100 ITALY
zimeo@unisannio.it

Abstract

Workflow enactment systems are becoming an effective solution to ease programming, deployment and execution of distributed applications in several domains such as telecommunication, manufacturing, e-business, e-government and grid computing. In some of these fields, efficiency and traffic optimization represent key aspects for a wide diffusion of workflow engines and modeling tools. This paper focuses on a technique that enables fine-grained concurrency in compute and data-intensive workflows and reduces the traffic on the network by limiting the number of interactions to the ones strictly needed to bring the data where they are really necessary for continuing the flow of computations. We implemented this technique by using the concepts of wait by necessity and automatic continuation and we integrated it in a flexible, Java workflow engine that through the new mechanisms is able to navigate a workflow anticipating the enactment of sequential activities.

1. Introduction

Workflow management is a field of growing interest thanks to the great flexibility and manageability ensured by the paradigm. It could be usefully employed in several domains to simplify the development and deployment of distributed applications characterized by common features like the access to computation resources scattered across a communication network. These applications are typically interested to access to the functionalities offered by remote resources, through a control flow that determines the sequence of actions to be performed to reach an objective and data to be transferred from a resource to another one [3].

In the domains characterized by compute and data intensive workflows, a process typically involves powerful computational resources used to compute complex functions and the operating data could require massive transmissions across the network. Therefore, any possibility to improve the performance must be carefully analyzed.

In this paper, we propose to relax the strong sequence constraint of the well-known sequence workflow pattern [2][13] at both control and data flow levels. In this pattern, the involved activities have to be executed in a serial way: the control is suspended until the first operation completes and only when the resulting data are send back to the workflow executor the process graph navigation can continue. Additional waiting time could be due to large data flows that require much time to be delivered, especially when data travel back and forth from the executor to the participants' resources several times during the enactment of a sequence of activities. Such a scenario does not benefit of the concurrency offered by the pattern "and-split" [2], since the activities present data dependencies and so they must be executed sequentially. Nevertheless, an improvement of efficiency is possible considering the problem at a different level of granularity, by dynamically finding the point inside the depending activity where the data produced by a previous activity are really needed. In particular, a potential concurrency could be exploited considering that the data dependencies of two sequential activities may not be placed at the beginning of the dependent activity. Data coming from an activity could be needed by the dependent activity after start-up and initialization operations. Therefore, we observe that between the two extreme cases of sequence and and-split there are different degrees of partial concurrency that could be exploited.

Considering a sequence of two activities, the corresponding control flow could be improved if we could start both of them at the same time, so the independent operations of the second activity may be executed concurrently with the ones of the first activity.

The implementation of this feature in a workflow enactment system can be performed by means of

1-4244-0054-6/06/$20.00 ©2006 IEEE

asynchronous invocations (or deferred synchronous invocations). In such a case the activity is started and the workflow executor receives a notification of the successful operation, so it could continue to navigate the workflow process graph until succeeding activities do not need data. To allow the continuation of the computational flow, the executor has to pass a symbolic reference of data to be computed. This reference represents a placeholder for the data that will be produced.

An additional benefit of the proposed schema is the ability to reduce traffic in the network since reference handling allows for avoiding data to be sent to the workflow enactment system and then to the next resource that requires them. Data could be transferred directly from the producer activity to the consumer one.

In [1], Manolescu has discussed the implementation of an object-oriented, micro-workflow enactment system that similarly to our proposal exploits asynchronous invocations and future objects. In addition to these basic ideas, our approach aims also at simplifying process modelling.

The rest of the paper is organized as follows. Section 2 discusses fine-grained concurrency as a useful pattern placed between sequence and parallel-split for designing workflows. Then in Section 3, an implementation based on future objects of the workflow fine-grained concurrency is presented. Section 4 discusses the results of some preliminary and basic experiments to test the efficiency and validity of the proposed solution. Section 5 concludes the paper and highlights future improvements.

2. Fine-grained concurrency in workflows

The objective of fine-grained concurrency is to exploit a great number of situations where computational flows can not be executed concurrently and then fall under the sequence way of enactment [3][8]. If the internal structure of an activity is considered, we can observe that data dependencies could appear in different points of the execution; so even though two activities can not run concurrently following the well-known and-split pattern [3] this is not sufficient to force a sequential execution since a partial concurrency is still possible. To this end, the key concepts that we utilize are anticipation and continuation [1], pointing at the removal of strict sequential execution of activity sequences in workflows by allowing intermediate results, or symbolic reference to them, to be used as preliminary inputs into succeeding activities thus enabling anticipation of enactment. This enables the continuation of workflow graph navigation, evaluating the possibility to start subsequent activities without waiting for the preceding to complete. To better explain the concept, we consider a simple sequential flow [13] of two activities **A** and **B**. The second one presenting a data dependency from the preceding one. The point of dependency could be placed in a whatever point of the second activity. As depicted in Figure 1, **B** could be considered as a sequence of two sub-activities: $\mathbf{B^i}$ and $\mathbf{B^{ii}}$. The first one could be interpreted as the subset of **B** operations that do not present data dependencies from **A**, while $\mathbf{B^{ii}}$ is the part that presents data dependencies. If during the analysis phase and subsequent design of a workflow the designers could examine the activities at a major level of detail, the different structure of the constitutive operations could be considered.

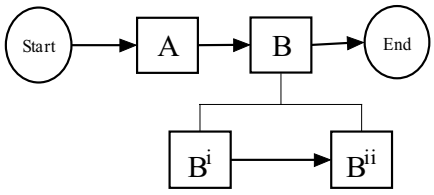


Figure 1. Sequential Flow

Taking into account such a structure, we could define an equivalent control flow, placing **A** and $\mathbf{B^i}$ in an and-split and then $\mathbf{B^{ii}}$ after the join of the two activities. The resulting control flow is depicted in Figure 2.

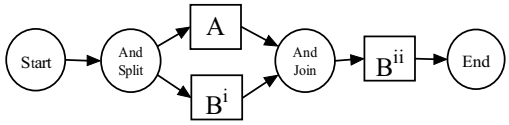


Figure 2. Equivalent flow considering a fine-grained analysis

Sub-activity $\mathbf{B^i}$ is enacted in parallel with **A** and, when the "and-join" [13] is completed , sub-activity $\mathbf{B^{ii}}$ is executed. By observing the process, the concept of fine-grained concurrency is used as a new control structure between the two extreme cases of sequence and and-split. The sequence is obtained if the $\mathbf{B^i}$ operations subset is void and $\mathbf{B^{ii}} = \mathbf{B}$, the and-split node is achieved if $\mathbf{B^i} = \mathbf{B}$ and $\mathbf{B^{ii}}$ is void. All the other intermediate situations could be more fittingly modeled with fine-grained concurrency whereas in the traditional workflow modeling they are described as sequences.

From a dynamic point of view the situation is presented in Figure 3, with the conceptual and simplified assumption that the outstanding time at stake are only the activities' duration time, giving up others times.

Figure 3.a shows the traditional way a workflow executor operates when the workflow presents a sequence. The activity **A** starts and runs until its operations are completed, then the result is returned to the workflow enactor and the activity **B** may be started receiving the result of **A** [2][3]. We are not considering dead time of the workflow system nor network delay. Figure 3.b shows the sequence enacted considering that the data dependency appears after the initialization activity. Both activities are started at same time. Activity **A** is invoked and returns a placeholder for the result of its computation. Activity **B** receives the placeholder as input data, runs until it reaches a point in its flow that requires the real data, tries to access to the placeholder to retrieve the result and, if it is not available yet, the activity is posed in suspend state, waiting for the result. When **A** completes its elaboration, the real data replaces the placeholder and **B** may restart completing its execution. As figure 3 shows, the second execution offers a gain equals to $\mathbf{T_\Delta}$.

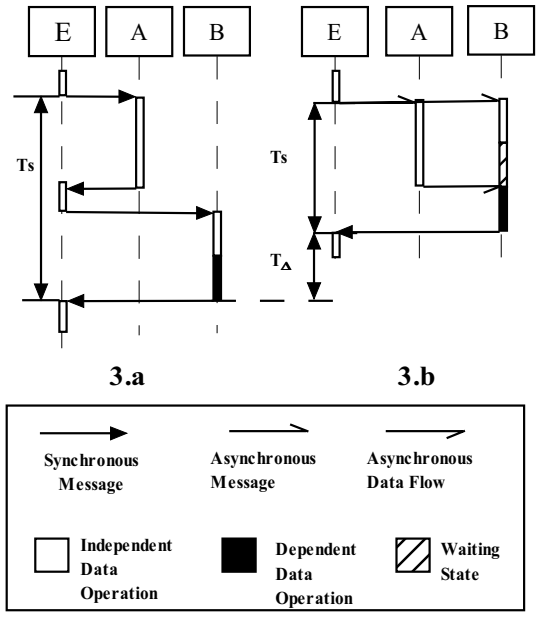


Figure 3. Comparison of sequence and fine-grained concurrency enactments

It is possible to prove that the gain is equals to the execution time of the **B** activity's operations that do not require the real data, i.e. $\mathbf{B^i}$. That operations could be executed concurrently with execution of activity **A**. Considering the point in the **B** activity's flow that notifies the data dependencies, it is possible to state that nearer the point is to the end of **B** greater the performance gain is (best case). Vice versa, if the point is nearer to the beginning of **B**, the gain is smaller (worst case). For long-time running operations the performance gain could be very considerable for this improvement in the control flow.

Since an arbitrary number of distinct split points could exist in real applications, in the different dynamic sequences defined by control flow of an activity, the dynamic approach represents the only viable solution to individuate such a point for complex activities and a useful support to alleviate the burden for designers regarding simple activities, where the split point could be individuated through a static analysis. Moreover, in some cases the internal structure of activities is not known by the workflow designer and so potential concurrency can not be exploited at design time. This limitation, induces designers to adopt the sequence pattern even in the very common cases where a partial concurrency could be exploited. However, the decomposition of all activities to statically find the dependence point it is not always a trivial task (if the source code of activity's implementation is available) or possible (if the source code is not available). The consequence is that any activity that presents a data dependence is treated as an activity composed of operations to be executed after the completion of the preceding activity and this is correct from a high-level point of view.

The classic sequence pattern, however, does not take care of the point in the subsequent activity that signals the real need for data. This may be done if we manage the activation at run-time of data dependent activities in a new way. All the activities in the sequence could be started as soon as possible, so they will execute concurrently up until the moment they really need the data from which they depend on.

The design phase of the sequence still is done in a traditional way, without forcing the designer to make assumption on the real data dependency, all the work is made by the system at run-time. To realize such a behavior the Workflow Enactment System must satisfies four requirements for anticipation and continuation that we defined:

1. Invocation of the activities must be done asynchronously, returning a symbolic placeholder for the result not computed yet.
2. Placeholder could be forwarded to subsequent activities as actual parameters to satisfy the activation conditions and so anticipating the activation.
3. Activities that receive the placeholder and try to access to the data must be stalled until the data is ready to be used.
4. The forwarded placeholder must be updated as soon as possible to each activity that uses it.

All these requirements must be accomplished in a completely transparent way, from the developers' point of view, so that traditional analysis and design can still be applicable.

If a workflow enactment system implements the requirements listed above, we could use continuation of workflow graph navigation and anticipation of activities enactment, achieving a great efficiency both in control-flow and data-flow. Continuation and anticipation must be performed at same time, because one requires the other. The accomplishment of the requirements by an enacting system could allow designers to limit their efforts to use a new workflow pattern, that we have called early start pattern, without performing a deep analysis of activities involved in a sequence. The process does not need to be rewritten, the dependence point is found dynamically by the system. The only burden for the designer is the declaration of optimization through the new pattern if a process definition tool is used or a language keyword if a process definition language is directly employed.

3. Implementation with Future Objects

The Workflow Engine (previously called enactor at high level), the software subsystem that navigates the process graph and enacts the activities, used in this work is part of an ongoing project, aimed to the definition and the implementation of a Workflow Enactment Service according to the SOA (Service Oriented Architecture) paradigm and able to select and bind the resources to execute the activities at run-time, starting from an abstract description of a process.

The implemented, experimental engine, called αEngine, is compliant with the Workflow Management Coalition (WfMC) abstract reference model [11], so it can execute a process defined by XPDL (XML Process Definition Language) [12] language and implements an API (Application Programming Interface) to interact with the other logical entities [11]. The participants of the workflow are treated as Invoked Applications representing services modeled as passive, local and remote objects, active, local and remote objects, and Web Services.

αEngine is able to navigate the process graph, the representation of the active instance of the process, and to choose the activities that could be executed according to the data contained in a centralized shared memory, it supports local or remote activities execution. The local execution requires introspection and reflection to perform the actions specified in the process. The remote execution is enabled by RMI (Remote Method Invocation) or SOAP (Simple Object Access Protocol), for traditional passive objects and Web Services.

To achieve anticipation and continuation, αEngine uses the asynchronous invocation (better known as deferred synchronous) of the remote activities implemented by the active objects provided by ProActive [10]. In such a way, the resulting system is able to satisfy the four requirements for anticipation and continuation introduces above.

ProActive is a pure Java library for parallel, distributed and concurrent computing, that provides a framework implementing the Active Object pattern [5]. An Active Object is represented as an object, with an interface exposing public methods, that have its own control thread. It is designed to improve simplicity and reuse in parallel programming, supporting separation of concerns related to functional and system aspects of programming [10]. ProActive allows for asynchronous calls by means of Future Objects [7][8].

A Future Object is an empty container that will receive the value resulting from an asynchronous method invocation [10]. Future Objects in ProActive are transparent because they do not require any modification of the caller's code.

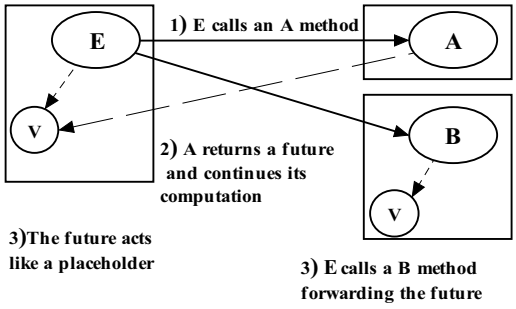


Figure 4. Continuation and anticipation with futures propagation

Figure 4 shows a typical interaction between active objects, in which **E** represents a workflow enactor and **A** and **B** two activities. The caller object **E** invokes a method of active object **A** and is not forced to wait for the method termination since a future of the value allows **E** to continue the execution of other statements.

If an object tries to access to the future, reading a properties or invoking a method, and the future object has not been substituted by the real object, the calling thread is blocked until the data is available. The future returned to **E** is updated by object **A** when its computation is completed. Furthermore, futures enable the continuation of the elaboration because they could be propagated to other activities outside the one that has received them. Propagation implies to pass the

future as a parameter or return it as a result without blocking [5][6].

When the actual value of the future is available, it is returned to the caller. Then it is propagated to all objects that received the future [10]. This strategy of updating is realized by a particular thread in the ProActive middleware that manages future objects. In this way, each active object is responsible for updating the futures it has forwarded.

In the basic implementation of the workflow engine, the asynchronous interactions between the enactor and the invoked applications are performed through two RMI synchronous invocations along with the implementation of the call-back pattern, which is used to notify the activity completion and the resulting data availability. However, in such architecture an activity containing an RMI invocation can not be considered completed until the engine will receive the call-back from the resource, so subsequent activities can not be analyzed and started, thus anticipation and continuation can not be performed.

In advanced implementation of the workflow engine, the asynchronous interactions between the enactor and the invoked applications are performed through ProActive. In this case, after the initial deferred synchronous invocation coming from the enactor, the invoked application returns a reference to the future resulting data of its computation, and starts the operation necessary to obtain it. Meanwhile, the engine receives the reference to the future result of the method, still to be computed: the placeholder. At this point the placeholder could be used to pass the virtual result to other applications. The only point of disparity with ProActive is on the future updating strategy that is not optimal for data flow improvement. We avoid such strategy by passing future objects by reference instead of by value. The future object is stored in the shared memory of the engine, while the anticipated activities receives a symbolic reference to the object as parameter instead of the real future object. The future is stored only in the engine shared memory and when data is available it replaces the future only there. If an activity reclaims the data, accessing to the central memory before it is available, the threads serving the request is stalled. When the data is available it is propagated only where it is needed. The stalling thread, in fact, is awaked and the real value is sent to the activity. At this point, when the central memory is accessed the real data is available. Following this scheme, the engine is not forced to update all the copies of the future previously forwarded. The updating happens on demand only when the activity really needs the data. This is useful to minimize data flow, but a further improvement is still possible as discussed thereafter. The central memory is used in our Workflow Engine to avoid a current inefficiency in ProActive updating strategy of future objects.

The implementation of continuation and anticipation into our engine has needed the introduction of a new state in the state diagram proposed by the reference model of WfMC [11]: *Semi-Complete*.

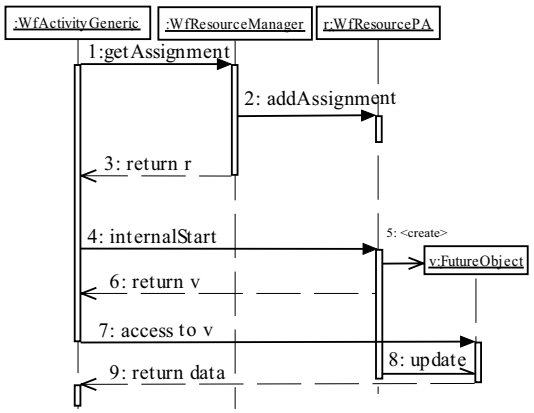


Figure 5: sequence diagram of αEngine implementation for remote services invocations

We need that the engine is aware of the situation that an activity may be still running to produce a real result, but nevertheless it is possible to continue analyzing the process graph. When the engine performs such an asynchronous invocation the state of concerning activity is switched from Running to Semi-Complete. This indicates the fact that while an activity is still running the engine could continue the process graph analysis to check if other activities could be enacted, i.e. anticipated. When the activation requirements are satisfied, also using the placeholder, the subsequent activity is started. The continuation of graph analysis goes on until activities can be anticipated, aiming to completely exploit fine-grained concurrency.

Figure 5 shows the interaction between the internal representation of activities (implemented according to the WfMC specifications) and external resources implemented by means of ProActive active objects.

4. Performance Evaluation

To validate the proposed technique, an experimental analysis was performed, using a simple workflow with two activities **A** and **B** in sequence, as depicted in Figure 1.

Activity **A** produces a data that will be used by activity **B**, that requires that data to perform its task. Activity **A** is a series of operations on a integer value, that require about 10 seconds to complete. Activity **B** receives the integer and performs a set of operations that also completes after 10 seconds. Thus the process under test is a sequence pattern with a data dependency in the second activity. The data-flow is less significant, since the integer is a very small data. This kind of process, enacted in a traditional way, starts activity **A**, then suspends the workflow enactment and waits until activity **A** completes. After that, the engine receives data from the resource and can use them to start the subsequent activity **B**. The total running time of the process is about the sum of the activities' running times, plus engine's enactment time and network's data transfer delay.

We examined the performance when the flow is executed with RMI remote passive objects and with ProActive remote active objects. In the second case, we considered the two extreme hypothesis: (1) data dependency is at the beginning of the activity; (2) data is needed in the end. The activities are compute-intensive applications that are long-time-running in respect to the total duration time of the process, because the elaboration time is greater than the engine operation time and the data transfer time. We conducted five executions of each kind of process, considering the average time as resulting performance indicator.

The computing resources used were:

PC1 – Intel Pentium 4, 2.0 MHz, 512 MB;
PC2 – Intel Pentium 4, 2.4 MHz, 512 MB;
PC3 – AMD Athlon XP 1800+, 1,53 MHz, 384 MB.

The experiments were executed with two different kinds of deployments, shown in Figure 6.

In figure 6.a, a computer runs the workflow engine and another one runs both the services encapsulated in the objects **A** and **B**. In figure 6.b, three different computers are used for the deployment: one for the engine, another to run activity **A** and the last to run activity **B**.

The obtained results are reported in table 1 where the execution time is measured in milliseconds. The process executed with RMI and normal sequence patterns runs in a total time slightly greater than the sum of the two activities running times. The first case, obtained by using ProActive, presents an improvement of performance of about 4% with respect of the basic execution based on RMI. However, this represents the worst case.

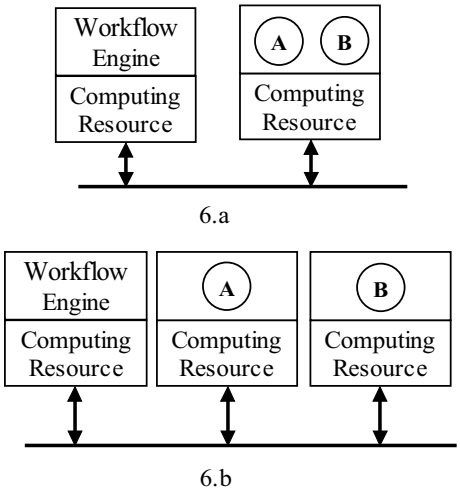


Figure 6. Deployment of activities on computing resources

The gain is due the fact that initialization operation of **B** could be conducted concurrently with **A** execution, thanks to the enactment anticipation performed by the engine. The latter case is the best possible for the engine. Here, the real need of the data is at the end of the activity, and thus a large part of independent computation could be performed concurrently to the execution of **A**, obtaining about 43% average better performance.

Table 1. Experimental results

Deployment	Services on the same computer Total time [ms]	Services on different computers Total time [ms]
RMI	20469,4	20505,8
ProActive Beginning Data Dependencies	19675	19784
ProActive Ending Data Dependencies	11647,4	11689,4

To understand the rational of the performance improvement obtained with our proposal, and at same time to better explain the dynamic behavior of the system, in the following we show a possible internal structure of the activities involved in the sequential sub-process employed for the experiment that we call *processAB*. This process could be sketched as shown in figure 7.

The execution of the two activities with the basic implementation of the engine proceeds as in figure 8.a, without early start pattern. The value needed by activity **B** could not be provided while **A** is still running. Using the advanced engine implementation, the execution proceeds as in figure 8.b. **A**, soon after the invocation,

returns a symbolic placeholder for its result. This placeholder is then passed to **B** as actual parameter (row 20). **B** could run until the instruction that needs the placeholder value is reached (row 12), at that point **B** is stalled, and awaked only when the real result has been updated (row 4).

```
1  IntWrp A() {
2    IntWrp i;
3    i = someProcessing();
4    return (i);
5  }
6
7  IntWrp B(IntWrp x) {
8    IntWrp j, k;
9    // Independent data operation
10    k = someProcessing();
11   // Dependent data operation
12    j = someProcessing(x, k);
13
14    return (j);
15 }
16
17 void processAB() {
18    IntWrp value;
19    value = A();
20    value = B(value);
21 }
```

Figure 7. Example code of two activities and a simple sequential process

The object IntWrp, is a wrapper used for the Integer due to some constraints imposed by ProActive on the methods result type.

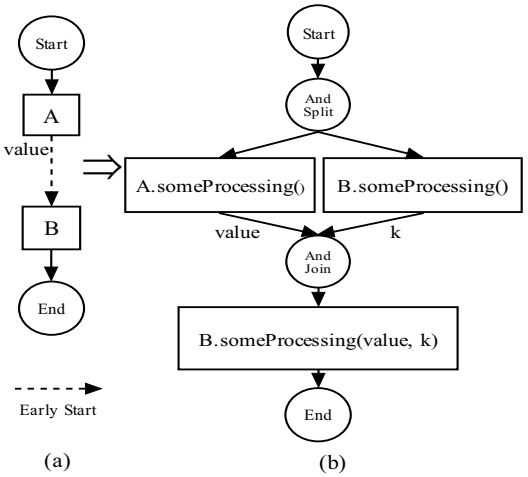


Figure 8. (a) Design of the workflow example with early start instead of sequence; (b) equivalent workflow with existing patterns

Summarizing, row 12 in activity **B** could be seen as the "dependence point". The preceding instructions are the independent data activities, the operation of **B** executed before the access to x parameter. The succeeding instructions, from row 12 to the end, are the dependent operations, i.e. they could be completed only after the successful access to the x parameter. ProActive allows to return a placeholder to processAB, that not suspend execution in row 19 but could go to row 20 and invoke B.

Other experiments confirm the above results. Using more complex workflows, presenting longer sequences, we observed the same average performance improvement.

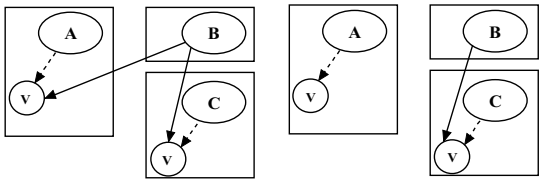
We stress the point that this performance is due only to control flow improvement, since data flow is the same in every case, i.e. data travel back and forth from engine to resource and vice versa. This is consequence of the shared central memory used to exchange data between the application and the on demand policy to update the future. This minimize the network occupancy but not optimize data-flow efficiency.

On the other hand, the currently implemented ProActive forward-based strategy to update the futures is not the best strategy for data-flow optimization because it requires that the caller is responsible for updating the values of futures it has forwarded. And this cause the sending of data to all the activities that received a future, without taking into account if they use the result or not.

Therefore, we schedule to employ an alternative strategy to improve data flow in our system. We are considering two strategies. Both of them are aimed to avoid to leave to the caller object the responsibility of updating all the forwarded futures. The called object, instead, must provide the real value for the future itself returned to the caller. We briefly describe the alternative strategies.

The Eager Message Based strategy, depicted in figure 9.a, implies that each forwarding of a future generates a notify message to the computing activity. This is responsible for sending the value to all the objects to whom the future was forwarded. The Lazy strategy, illustrated in fig 9.b, updates the future only when its value is effectively used in computation. When an object effectively tries to access to a future, if the data is ready the producer updates the future value. Both solutions could be useful, because them brings the data to the subsequent activity in the sequence without involving the engine.

We think that the Lazy strategy is the best solution. The reason is bounded with the nature of workflow data. They could be Workflow Relevant Data and Workflow Application Data. The formers needs to be processed by the engine because they are used to determine the next path to follow in the process graph.



9.a 9.b

Figure 9. Proposed futures' updating strategies

The latter are not needed by the engine but only by the participants. The Lazy strategies ensures two advantages: first of all, data is delivered to the resource that needs it, when necessary, directly from the producer activity, without passing across the engine, reducing the data transmission and shortening the delivering delay. The second accomplishment is due the fact that the data are delivered to the engine only if they are Workflow Relevant Data, because the engine try to use them. This is an important result because it is possible to avoid unnecessary data transmission without requiring to specify the different nature of the data at design time, but everything is made transparently at run-time.

5. Conclusions

This paper discussed the improvements that a workflow management system could achieve if activities are not considered as atomics, but taking into account that they have an internal structure composed of several operations that typically are not accessible or visible. This suggested us to introduce workflow fine-grained concurrency, that could allow partially concurrent execution of apparent sequential activities. Basing on this concept, the paper described continuation and anticipation in workflow enactment, that together maximize performance improvements as demonstrated by the performance evaluation section.

The current implementation of the engine exploits control flow improvements, but is not able yet to gain full advantage of all the potential data flow optimization. Therefore, anticipation and continuation improve control-flow management since they do not require particular care during the analysis and design processes. However, the engine should be further improved taking into account data flow optimization: data could be transferred to the place where they are effectively needed to allow computation to going on, avoiding to be sent back and forth on the communication media.

It is worthy to note that the improvements described are strictly bound to the position of the split point in dependent activities. This could open a new horizon to the employment of code optimization and reordering, to achieve best performance at run time.

6. Acknowledgements

This work is framed within the project LOCOSP n° 4452/ICT funded by MIUR – Italian Ministry of University and Research.

References

[1] Dragos-Anton Manolescu, "Workflow Enactment with Continuation and Future Objects", Proceedings of OOPLSA'02, Seattle, WA, 2002.

[2] Bartek Kiepuszewski, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, "Fundamentals of control flow in workflows", Acta Inf. 39(3), 143-209, 2003.

[3] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, Alistair P. Barros, "Workflow Patterns", Distributed and Parallel Databases 14(1), 5-51, 2003.

[4] Wil M. P. van der Aalst, Alistair P. Barros, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, "Advanced Workflow Patterns", CoopIS 2000, 18-29, 2000.

[5] Laurent Baduel, Francois Baude, Denis Caromel, "Object-Oriented SPMD", Proceedings of Cluster Computing and Grid, Cardiff, United Kingdom , 2005.

[6] Denis Caromel, Ludovic Henrio, Bernard Paul Serpette, "Asynchronous and Deterministic Objects", POPL 2004, 123-134, 2004.

[7] Denis Caromel, "Towards a Method of Object-Oriented Concurrent Programming", Communication of ACM 36(9), 90-102, 1993.

[8] Denis Caromel, Wilfried Klauser, Julien Vayssière, "Towards Seamless Computing and Metacomputing in Java", Concurrency - Practice and Experience 10(11-13), 1043-1061, 1998.

[9] D. Georgagopoulos, M. Hornick, A. Sheth: "An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure", Distributed and Parallel Databases, 3, 119-153, 1995

[10] ProActive Manual, www-sop.inria.fr/oasis/ProActive/.

[11] Workflow Management Coalition, "The Workflow Reference Model", Document Number WfMC TC-1003, www.wfmc.org.

[12] Workflow Management Coalition, "XML "Process Definition Language", Document Number WfMC TC-1025, www.wfmc.org.

[13] Workflow Management Coalition, "Terminology and Glossary", Document Number WFMC TC-1011, www.wfmc.org.