Workflow Partitioning to Support Decentralised P2P Based Enactment

Tariq N. Ellahi, Francesco Nerieri and T. Kechadi

School of Computer Science and Informatics
University College Dublin
Dublin - Ireland
{tariq.ellahi, tahar.kechadi}@ucd.ie, nero@google.com

Abstract. Grid workflow management systems has been a focus of research in recent years and as a result a lot of systems have been developed. Most of the existing systems employ a centralised workflow enactment engine to coordinate the execution of workflow activities. This approach results in a single point of failure and increased execution overhead due to the existence of a single enactment engine. In this paper, we present a workflow partitioning mechanism which can enable decentralised enactment of the workflow activities by a number of distributed workflow engines coordinating in a peer-to-peer fashion. Initial Experimental results show the efficiency of the partitioning procedure.

1 Introduction

Grid computing [1] enables the aggregated utilization of heterogeneous and distributed resources such as computing elements, data sources, instruments and application services. With the help of grid systems, the scientific community is developing and deploying complex applications to manage and process large data sets, and execute scientific experiments on distributed grid resources. A workflow application is a collection of computational activities to be executed according to the user-specified invocation order to accomplish some specific goal. Workflow application is represented as a directed graph. Activities are modeled as vertices of the graph and dependencies among activities are represented as edges. Workflow management systems define, create and manage the execution of workflow applications through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications. Workflow management systems have been used by the business community to manage business processed for a very long time. Recently, workflow support in the grid systems was adopted and a number of grid workflow management systems have been developed during the last few years.

The existing grid based workflow management systems employ a centralised workflow enactment mechanisms. In this scenario, a single workflow execution engine coordinates the execution of all the activities in the workflow graph. This architecture not only introduces a single point of failure but it severely restricts

Advances in Computer Science and Engineering
Research in Computing Science 27, 2007, pp. 291-304

Received 13/02/07
Accepted 08/04/07
Final version 23/04/07

the scalability of the system especially in the grid landscape where thousands of users will be using the system. One solution to overcome this problem is to distribute the workflow enactment tasks among a number of workflow engines. A couple of grid based workflow management systems adopt the decentralised enactment approach. In order to be able to support the decentralised enactment of workflow applications, workflow graph must be partitioned and each partition is assigned to a different workflow engine. This approach will improve the scalability of the system and result in increased efficiency and reduced execution time.
In this paper we present a workflow partitioning mechanism that will enable the grid system to execute the workflow application without relying on a centralized point of command and control. This approach is used in the grid workflow management system which is part of the Data Grid Environment & Tools (DGET) [2][3][4] grid middleware. The rest of the paper is structured as follows: Related work on workflow partitioning is discussed in section 2. Sections 3 and 4 present general information about workflow terminology and the lifecycle of a workflow application. Detailed explanation of the workflow partitioning procedure with examples are given in section 5. Analysis and evaluation of the workflow partitioning procedure is explained in section 6. Section 7 concludes the paper and presents future research direction.

2 Related Work

There are several workflow management systems that has be developed during last few years [5][6][7][8][9][10][11][12]. Almost all of the systems referenced except[5][6] employ centralised workflow enactment and thus no partitioning procedures are applied. There are a couple of workflow management systems that have been developed with the same objectives as our systems i-e decentralised enactment of the workflow application. Workflow partitioning was applied in the Pegasus [5] system but the objective and the method used were different than our partitioning procedure. The objective of the workflow partitioning system in Pegasus was efficient mapping of the workflow activities onto grid resources. The Workflow application is partitioned by levels of the graph thus each level was isolated as a separate partition. This approach didn't take into account the data dependencies among the activities which is a crucial factor in order to enable the system to support decentralized workflow enactment.

The second workflow management system that incorporates workflow partitioning is Askalon [13]. In Askalon partitioning is performed as an optimization of the workflow execution. Contrary to Pegasus and our system, the partitioning procedure is applied after the scheduling of the workflow activities has been performed. Its partitioning procedure is quite straight forward in Askalon. All the activities that have been scheduled for execution on the same grid site are put into one partition. This approach while useful to some extent still might not result in efficient enactment of the workflow application. Workflow partitioning before the scheduling process can take into account the data dependencies

among workflow activities. Thus different scheduling strategies can be applied to different partitions based on the nature of the activities contained in the partition.

3 Workflow Lifecycle

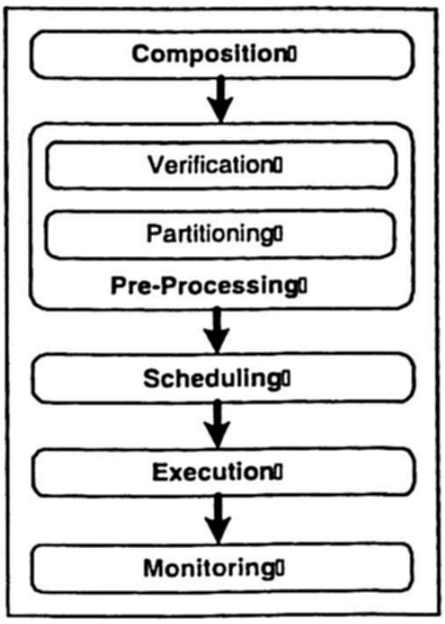

Fig. 1. Workflow Lifecycle

In this section we describe the typical lifecycle of a workflow application. Every workflow application passes through a number of phases during its lifetime. There are five such phases in the workflow application lifecycle. Following subsections explain each of the phase in details.

3.1 Composition

Workflow lifecycle starts with the composition of the workflow application using the workflow specification language or other high level visual composition tools. User specifies all the activities at an abstract level without specifying the grid related details. Beside specifying the activities, user also defines the dependency relations among the activities. After the workflow specification, the workflow is parsed and an internal java representation of the workflow graph is generated by the workflow engine. This java representation is used during the later phases of the workflow lifecycle. The workflow parsing process can also be taken as a pre-verification procedure where workflow is checked for errors missing values in the workflow specification.

3.2 Pre-Processing

After the composition phase, workflow graph is pre-processed. The pre-processing phase is composed of two sub-phases: Verification & Partitioning. These sub-phases are explained in the following paragraphs.

Verification As described in the previous section, workflow specification is parsed and checked for syntactical anomalies in the composition phase. In the verification phase, workflow graph is checked and verified for any structural conflicts like deadlocks, livelocks and lack of synchronization etc. The verification procedure is explained in a separate section later in this paper.

Partitioning After the successful verification of the workflow graph, the partitioning phase is entered. Since the focus of our work is peer-to-peer based enactment of the workflow, workflow specification must be decomposed into a set of distinct activities that can be enacted by a number of cooperating workflow engines. This decomposition of done in the partitioning phase. Section * describes the partitioning procedure at great length.

3.3 Scheduling

Workflow application specified during the composition phase did not include any grid related information e.g resource which are going to be used to execute the activities. During the scheduling phase, the activities are mapped onto grid resources taking into account the QoS parameters specified in the activity definition. The scheduling is performed at the partition level. Services of middleware components like Resource Discovery are used to find matching resources that can be used for activity execution.

3.4 Execution & Monitoring

After successful scheduling of workflow activities, the execution of the workflow activities commence. Along with the execution of the activities, monitoring of the execution is also performed. if anything goes wrong with the execution of some activity, the monitoring component triggers the fault management component so the execution rescue can be performed.

4 Workflow Partitioning

Workflow partitioning is the process of decomposing the workflow graph into a number of smaller non-empty, non-overlapping set of activities. Different partitions can be assigned to different workflow engines so that workflow can be executed in a distributed and decentralized fashion. Partitioning procedure takes into account the minimization of the inter-partition dependencies. This results in a number of distinct advantages:

- Distributed and decentralized workflow enactment in a Peer-to-Peer fashion. This also eliminates a centralized workflow engine overlooking and coordinating the execution of overall workflow application.
- Reduced Scheduling and job submission overhead
- Reduced Communication Overhead
- Increased fault tolerance and resilience
- Improved execution efficiency

4.1 Notations

The following notations will be used in this document to explain different aspects of workflow management.

- $\mathcal{W}$: Workflow graph with vertices representing the activities and edges representing the data dependencies between the activities
- $\mathcal{A}$: Set of activities in the workflow $\mathcal{W}$ (Vertices of the graph)
- $\mathcal{D}$: Data Dependencies between activities in the workflow $\mathcal{W}$ (Edges of the graph)
- $\mathcal{P} = \{p_1 \dots p_n\}$: Set of partitions of the workflow $\mathcal{W}$
- $a_i \longrightarrow_d a_j$: Data dependency between activity a_i and a_j where $a_i, a_j \in \mathcal{A}$
- *pred*(a): Direct predecessor(s)/parents of an activity a $\in \mathcal{A}$
- *succ*(a): Direct successor(s)/children of an activity a $\in \mathcal{A}$
- s: Start activity of the workflow $\mathcal{W}$
- s_p: Start activity of a partition p $\in \mathcal{P}$
- *level*(a): Level of the activity a in the workflow $\mathcal{W}$
- *ldiff*(a_i, a_j) = *level*(a_j) - *level*(a_i): Level Difference between activities (a_i) and (a_j)

4.2 Partition Properties

All the partitions in the workflow exhibit the following properties:

- Non-Empty: $\forall p_i \in \mathcal{P}, \mid p_i \mid > 0 \wedge \sum_{i=0}^{n} \mid p_i \mid = \mid \mathcal{A} \mid$
- Non-Overlapping: $\forall p_i, p_j \in \mathcal{P}, p_i \cap p_j = \emptyset \wedge \bigcup_{i=1}^{n} p_i = \emptyset \wedge \bigcup_{i=1}^{n} p_i = \mathcal{A}$
- No Intermediate Data Dependencies:

$$\forall a_i, a_j \in \mathcal{A}, \nexists\ a_i \longrightarrow_d a_j \text{ where } a_j \in p_i \in \mathcal{P} \wedge a_i \notin pi \wedge a_j \neq s_{pi}$$

4.3 Partitioning Algorithm

The general motivation behind the workflow partitioning is to divide the workflow activities into a number of disjoint sets of activities. By doing so, each subset of activities can be assigned to a separate workflow engine for enactment and therefore eliminating the existence of a centralized workflow engine overseeing the execution of overall workflow application. One critical factor in the partitioning is to make sure that such isolated activities in a partition must have

minimum possible dependencies and interaction with the rest of the activities in other partitions. Activities in a workflow applications are organized into different patterns, therefore, the functionality of the partitioning algorithm is to search for a number activities arranged in some patterns with least possible synchronization points. Before we describe the partitioning algorithm, it will be necessary to explain the different patterns, our algorithm will be looking for in the workflow specification. Partitioning algorithm will try to identify the following types of patterns:

Activity Pipeline: Activities arranged in a sequential spatial order. This pattern is similar to an execution pipeline where activities are started one after another without interacting with the rest of the activities in the workflow. Partitioning algorithm looks for a set of activities which are arranged in a sequential order with no synchronization points during the execution of the pipeline. However, pipeline can begin with a synchronization point which mean, the workflow engine will have to wait till the synchronization is achieved and after that, it can continue executing the rest of the activities in the pipeline without reliance on other activities in the workflow. Figure 2 shows an example of a pipeline, example on the left side show a pipeline without any synchronization point and the one on the right with a synchronization point at the beginning of the pipeline.

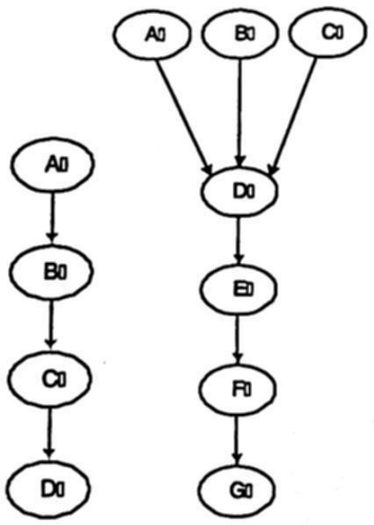

Fig. 2. Activity Pipeline

Parallel Block: The second type of activity pattern the partitioning algorithm tries to identify is a set of activities which can be executed independently of each other and from the rest of the activities in the workflow. Such activities exist when the execution flow reaches a split point and a number of activities are started that can be run in parallel. Different execution flows started by a split point will have to converge/synchronize at some stage later during the execution of the workflow. There can be different execution flow scenarios between a split

point and joint point. We identify the following scenarios and our algorithm treats each scenario in a different fashion.

Immediate Synchronization: This is the case where multiple instances of the same activity are started to achieve SPMD like scenario. All the instances synchronize at the next level. Another example could be where a number of different activities are started at the same time and they synchronize at the next level. Activities in this scenario can be isolated in a partition and executed by a separate workflow engine. Figure 3a is an example of this pattern.

Delayed Synchronization: In this scenario, the split point triggers a number of different execution flows that involves executing multiple activities in a pipeline or further split points can occur. These different execution flows converge at a later stage. The partitioning algorithm tries to find different patterns between the split and the join point and isolate them in different partitions. Figure 3b is an example of this pattern.

Hybrid: In this scenario, split point can instantiate a number of instances of the same activities and some execution flows like pipeline. The algorithm isolates the parallel activities in one partitions and the pipeline can result in a separate partition. Figure 3c is an example of this pattern.

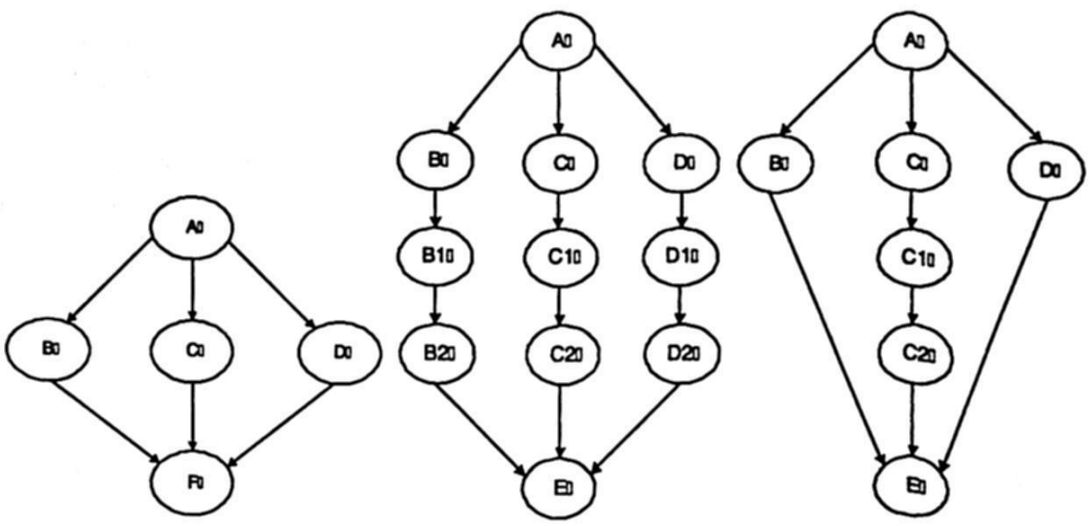

Fig. 3. Parallel Acivities

Iteration Block: Most of the workflow models are based on Directed Acyclic Graph (DAG) model. In such models loops are not allowed. There can be scenarios where some activity needs to be executed iteratively, therefore, our specification model allows loops in the workflow specification. Activities in an iteration block are isolated in a separate partition. Splitting activities in a loop into multiple partitions will result in dependencies among partitions, therefore, a separate partition is created for each loop definition specified in the workflow model. Iteration blocks are identified with the help of language constructs. This is done

before the partitioning algorithm is applied on the workflow specification. Figure 4 gives and example of iteration blocks both in graphical format and in XML format.

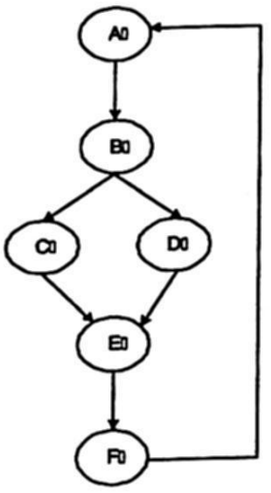

Fig. 4. Iteration Block

Conditional Execution: Another special case in the workflow specification is the conditional execution of the activities. In this execution splits into one of many possible execution flows depending on some conditional expression evaluated during runtime. As with the iteration blocks, conditional execution blocks are isolated into a partition of their own. Conditional execution blocks are identified with the language constructs from the workflow specification model before applying the partitioning algorithm.

Partitioning algorithm is given below:

```
PROCEDURE Partition (W)
QUEUE = {s}
while QUEUE ≠ ∅ do
  BLOCK = {}
  Get ν, First Node in the queue;
  if ν instanceOf Partition then
    remove ν from QUEUE;
    append succ(ν) to QUEUE;
  else if | succ(ν) | = 1 && ldiff((ν), succ(ν)) = 1 && | pred(succ(ν)) | = 1
  then
    remove ν from QUEUE;
    append ν to BLOCK;
    CALL extractPipeline(succ(ν));
  else if | succ(ν) | > 1 AND | pred(ν) | > 1 then
    remove ν from QUEUE;
    append succ(ν) to QUEUE;
```

```
    replace ν with a Partition
  else
    CALL extractParallel(ν);
  end if
end while
END Partition

PROCEDURE extractPipeline(startNode κ)
if κ instanceOf Partition then
  append succ(κ) to QUEUE;
else if | succ(κ) | = 1 AND | pred(κ) | = 1 then
  append κ to BLOCK;
  CALL extractPipeline(succ(κ));
else if | succ(κ) | > 1 AND | pred(κ) | = 1 then
  append κ to BLOCK;
  if succ(κ) NOT IN QUEUE then
    append succ(κ) to QUEUE;
  end if
  returnAFG(x + y)/2
else if | succ(κ) | = 1 AND | pred(κ) | > 1 then
  if κ NOT IN QUEUE then
    append κ to QUEUE;
  end if
  return
end if
replace nodes in BLOCK as a Partition in W
END ExtractPipeline

PROCEDURE extractParallel(startNode κ)
tempBlock = {};
append κ to tempBlock;
if succ(κ) NOT IN QUEUE then
  append succ(κ) to QUEUE;
end if
let δ = pred(κ);
while TRUE do
  ω = next node in the QUEUE;
  if pred(ω) ≠ δ then
    break;
  else if | succ(ω) | = 1 && ldiff((ω), succ(ω)) = 1 && | pred(succ(ω)) | = 1
  then
    remove ω from the QUEUE;
    CALL extractPipeline(ω);
  else
    remove ω from the QUEUE;
```

append ω to tempBlock;
append $succ(\omega)$ to the QUEUE;
end if
end while
replace nodes in the tempBlock as Partition in $\mathcal{W}$
return
`END extractParallel`

4.4 Partitioning Example:

Figure 5 and 6 gives two example workflow and their partitioned versions. Workflow graph structure in figure 5 is protein annotation workflow[14] and figure 6 is a neuro-science workflow[15]. The reason for taking these two as examples is because these workflow structures represent two different type of workflow graphs. Workflow in figure 5 is a regular workflow graph with regular data dependencies but the workflow graph in figure 6 is an irregular workflow with arbitrary data dependencies.

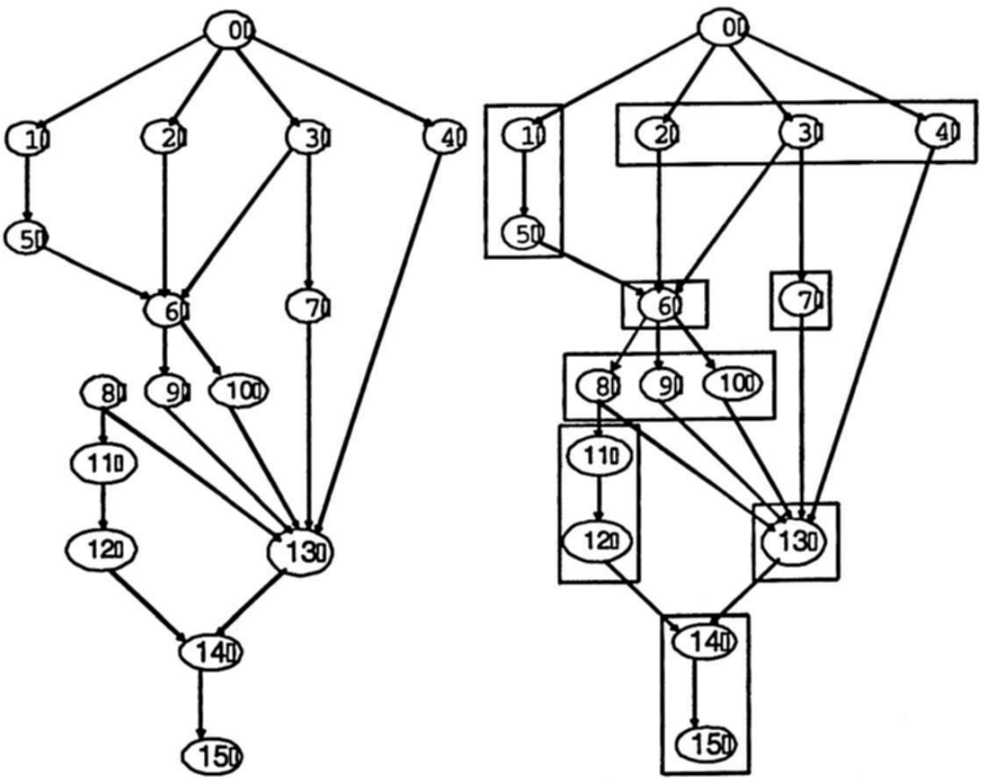

Fig. 5. e-protein Workflow Example

5 Analysis & Evaluation

This sections focuses on the analysis and evaluation of the workflow partitioning procedure. Initial experimental results are presented. Table1 shows the experimental setup and the machines used to run the experiments. Figure 7 and 8

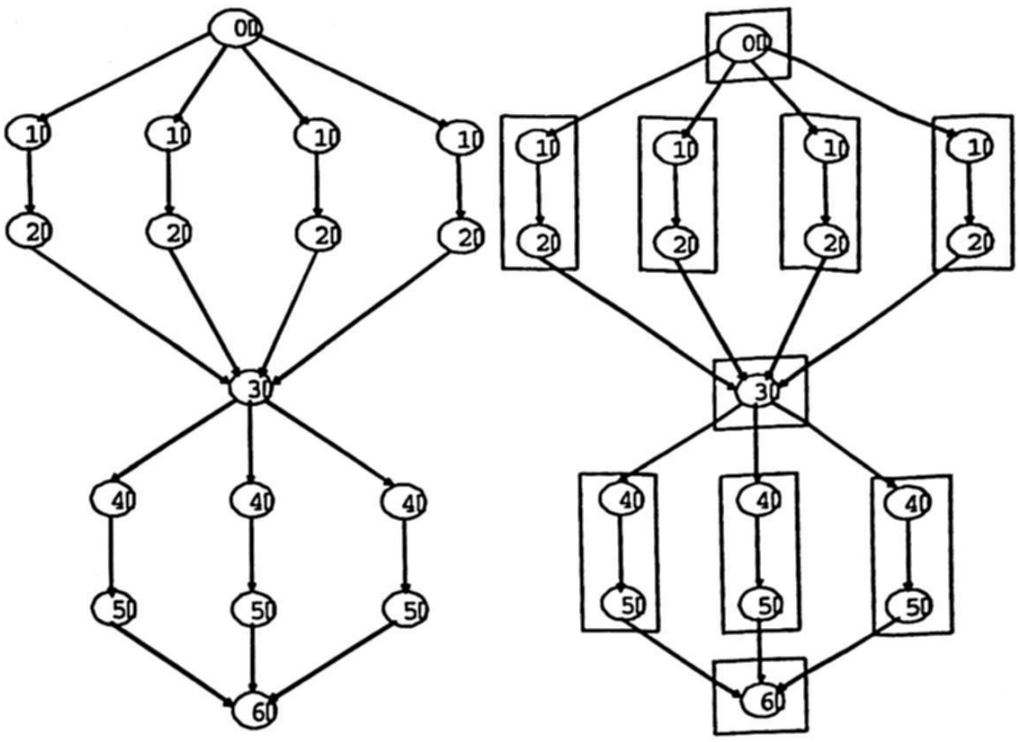

Fig. 6. fMRI Workflow Example

Table 1. Experimental Setup

Nr.	Machine	CPUs
1	bonnat.ucd.ie	4
2	pcrgcluster.ucd.ie	8
3	radostina.ucd.ie	2
4	bwhdzia.ucd.ie	1
5	tellahi.ucd.ie	1

shows the impact of partitioning process on the overall execution of the workflow application. Workflow partitioning doesn't add much overhead to the execution time of the workflow execution. As can be seen from figure 7, the partitioning phase constitutes 6% of the entire execution time of the workflow application but this cost of execution time spent in partitioning has its influence on the execution speed of the workflow application. Figure 8 depicts the execution times of a workflow application both in partitioned and non-partitioned scenarios. As can be seen the non-partitioned case takes more time to complete where the execution time is reduced to almost half when partitioning is performed. The overhead comes in the time spent to coordinate the execution of workflow activities by a single centralized entity. All the coordination messages has to be routed through the centralized enactment engine therefore increasing the execution time of overall workflow application.

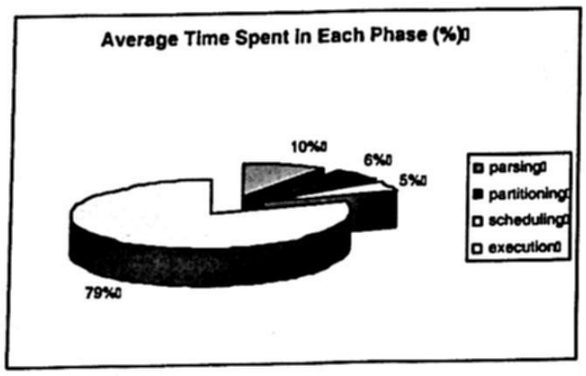

Fig. 7. Execution Time in Lifecycle Phases

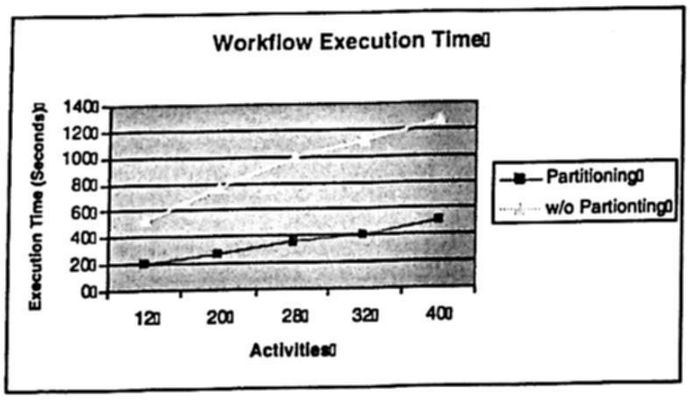

Fig. 8. Workflow Execution Time

Figures 8 and 9 depicts the partitioning behaviour when applied on different types of workflow graphs. Figure 8 represents the partitioning behaviou on regular and irregular workflow graphs. Since there are arbitrary data dependencies

in irregular workflow graphs, this results in a larger number of partitions being created especially some partitions where there is only one activity. This activity is typically an activity which has random data dependencies and can not be included in a serial or parallel block. In the sample workflow graphs, the number of activities were approximately the same but the structure of graphs were completely different. As a result, a regular graph is partitioned into half the number of partitions compared to the irregular workflow graph.

Figure 9 shows the partitioning behaviour on workflow graphs with majority of activities arranged in pipeline or parallel layouts. A workflow graph with majority of activities existing in pipeline form results in more number of partitions because activities in a pipeline can not have data dependencies on activities in other partitions. Whereas, in the second case, parallel activities do not have any impact and there are less number of partitions in the second case.

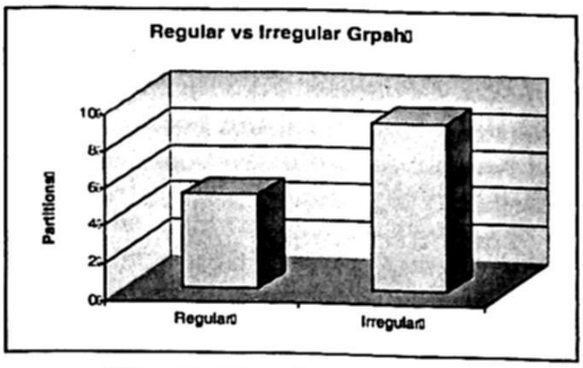

Fig. 9. Regular vs Irregular Workflow Partitioning

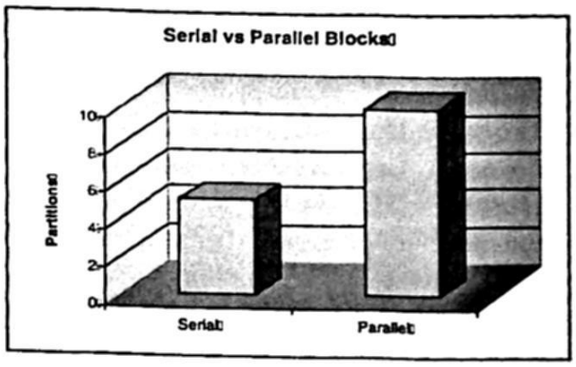

Fig. 10. Serial vs Parallel Block Partitioning

6 Conclusion & uture Work

In this paper we presented a workflow partitioning mechanism which can be used to split a workflow graph into a number of partitions. Each partition can be executed by a different workflow enactment engine. This approach eliminates a central entity coordinating the execution of entire workflow application. Experimental results show that partitioning a workflow application results in improved efficiency and reduced execution time for the workflow application. In the future, we would like to investigate the partitioning of different types of very large scale workflow involving thousands of activities. Another important research direction could be to ascertain whether partitioning is useful in all scenarios.

References

1. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure.* Morgan Kaufmann Publishers, USA, 1999.
2. L. McDermott T. Kechadi T. N. Ellahi, B. Hudzia. A java based architecture of p2p-grid middleware. In *The 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2006)*, 2006.

3. L. McDermott T. Kechadi T. N. Ellahi, B. Hudzia. Entity management and security in p2p grid framework. In *7th International Conference on Computing (CORE 2006)*, 2006.
4. T.N. Ellahi B. Hudzia, L. McDermott and T. Kechadi. Entity based peer to peer in data grid environments. In *17th IMACS World Congress, Paris, France*, 2005.
5. Mei-Hui Su James Blythe Yolanda Gil Carl Kesselman Gaurang Mehta Karan Vahi G. Bruce Berriman John Good Anastasia Laity Joseph C. Jacob Daniel S. Katz Ewa Deelman, Gurmeet Singh. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13-3:219 – 237, 2005.
6. Juergen Hofer Farrukh Nadeem Francesco Nerieri Stefan Podlipnig Jun Qin Duan Rubing Mumtaz Siddiqui Hong-Linh Truong Alex Villazon Marek Wieczorek Thomas Fahringer, Radu Prodan. Askalon: A development and grid computing environment for scientific workflows. In *Workflows for eScience, XXX (Eds.), Springer*, 2006.
7. Jia Yu and Rajkumar Buyya. A novel architecture for realizing grid workflow using tuple spaces,. In *5th IEEE/ACM International Workshop on Grid Computing (GRID 2004, Nov. 8, 2004, Pittsburgh, USA), IEEE Computer Society Press*, 2004.
8. Francine Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-Crummey, Dan Reed, Linda Torczon, and Rich Wolski. The GrADS Project: Software support for high-level Grid application development. *The International Journal of High Performance Computing Applications*, 15(4):327–344, 2001.
9. Junwei Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. Gridflow: workflow management for grid computing. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid CCGrid 2003*, pages 198–205, 2003.
10. Chad Berkley Dan Higgin Bertram Ludäscher, Ilkay Altintas. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows*, 2005.
11. Matthew Addis M. Nedim Alpdemir Justin Ferris Kevin Glover Carole Goble Antoon Goderis Duncan Hull-Darren Marvin Peter Li Phillip Lord Matthew R. Pocock Martin Senger Robert Stevens Anil Wipat Tom Oinn, Mark Greenwood and Chris Wroe. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, Volume 18 Issues 10:1067 – 1100, 2005.
12. David Churches, Gabor Gombas, Andrew Harrison, Jason Maassen, Craig Robinson, Matthew Shields, Ian Taylor, and Ian Wang. Programming Scientific and Distributed Workflow with Triana Services. *Concurrency and Computation: Practice and Experience (Special Issue: Workflow in Grid Systems)*, 18(10):1021–1037, 2006.
13. Thomas Fahringer Rubing Duan, Radu Prodan. Run-time optimization for grid workflow applications. In *7th IEEE/ACM International Conference on Grid Computing (Grid'06), (C) IEEE Computer Society Press*, September, 2006.
14. Darlington J O'Brien A, Newhouse S. Mapping of scientific workflow within the e-protein project to distributed resources. In *UK e-science all-hands meeting, AHM 2004, Nottingham, UK*, 2004.
15. Yong Zhao, Michael Wilde, Ian Foster, Jens Voeckler, Thomas Jordan, Elizabeth Quigg, and James Dobson. Grid middleware services for virtual data discovery, composition, and integration. In *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*, pages 57–62, New York, NY, USA, 2004. ACM Press.