J Supercomput (2013) 63:919–945
DOI 10.1007/s11227-012-0837-z

A model, design, and implementation of an efficient multithreaded workflow execution engine with data streaming, caching, and storage constraints

Pawel Czarnul

Published online: 21 November 2012

Abstract The paper proposes a model, design, and implementation of an efficient multithreaded engine for execution of distributed service-based workflows with data streaming defined on a per task basis. The implementation takes into account capacity constraints of the servers on which services are installed and the workflow data footprint if needed. Furthermore, it also considers storage space of the workflow execution engine and its cost. Caching service output data is implemented to speed up the execution of the workflow. Input data is partitioned into data packets, which are passed and processed by services previously selected for workflow tasks so that the aforementioned constraints are met. Performance impact of the proposed mechanisms is investigated for workflow structures common in acyclic directed graph workflow applications. It is shown for a real workflow with distributed processing of digital media content that the initial budget needs to be properly distributed between both the cost of services, but also the cost of intermediate storage to obtain good workflow execution times.

Keywords Workflow execution · Data streaming · Storage constraints · Service selection

1 Introduction

Integration of distributed services and applications is one of the main challenges in today's distributed systems. This applies to all distributed software architectures [7], in particular Service Oriented Architectures (SOA), grid [13], cloud [17], and sky computing [15]. Such integration is often modeled using workflow applications in

P. Czarnul (✉)
Department of Computer Architecture, Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Narutowicza 11/12, 80-233 Gdansk, Poland
e-mail: pczarnul@eti.pg.gda.pl

SOA, grid processing [9, 23, 27, 28], and recently also in cloud computing [17]. In this case, a workflow is defined as an acyclic directed graph in which each node denotes a task to be executed while directed edges denote dependencies between the tasks. For each task, a set of services capable of executing the given task is found first. Each workflow task is assigned a list of services each of which can perform the task on different Quality of Service (QoS) terms such as the execution time, cost, reliability, etc. The usually considered workflow scheduling problem is to select such a service for each task so that one of given global QoS goals is optimized [23] while other QoS conditions are satisfied. One of the most often optimized goals is minimization of the workflow execution time with a bound on the total cost of selected services [10, 26].

The outline of the paper is as follows. Section 2.1 presents models and algorithms for workflow scheduling in distributed service based systems. The available solutions related to storage-aware workflow management are discussed in Sect. 2.2. Section 2.3 lists motivations that are subject of analysis, proposal of a model and subsequent implementation and tests. Section 3 presents the proposed extensions to existing approaches with a formal definition of the model. Section 4 discusses design and implementation details of the multithreaded storage-aware workflow execution engine with data caching implemented by the author in the BeesyCluster middleware. A proposal of a genetic-based algorithm for workflow scheduling considering storage constraints is shown in Sect. 4.3. Section 5 contains a series of experiments that demonstrate the impact of BeesyCluster storage size used for storing intermediate data on the workflow execution time (Sect. 5.2.1), impact of multithreaded copying, caching, and storage constraints on the workflow execution time (Sect. 5.2.2) and various allocations of the initial budget among the cost of services and storage costs to obtain shortest possible execution time (Sect. 5.3). Finally, Sect. 6 contains conclusions and areas for future work.

2 Related work and motivations

2.1 Workflow application scheduling problem

Most of the literature on scheduling workflow applications considers the classic problem [28] in which for each workflow task t_i (represented by a vertex of an acyclic directed graph where the edges define dependencies between tasks) there is a set of services S_i each of which is capable of executing the task. Each task is supposed to process data of size d_i which is defined a priori. The goal of scheduling is to find mapping $t_i \rightarrow (s_{ij}, t_{ij})$ where t_{ij} is the time when the selected service s_{ij} starts processing data of size d_i (that must have been previously copied to the service). The criterion is to minimize the workflow execution time (the time when the last service has finished processing its data) while meeting $\sum_{i:s_{ij} \text{ selected for } t_i} c_{ij} d_{ij} \leq B$ where B is the available budget and d_{ij} is the size of data processed by service s_{ij}. Alternatively, one may want to minimize the total cost of selected services and ensure that the workflow execution time is below a given deadline. Various heuristic methods

have been proposed to solve these problems:

(1) A genetic algorithm [27] where a chromosome represents which services are selected for a particular task. Evaluation of each chromosome is performed to compute its fitness function (workflow execution time with a check whether it meets the cost constraint). Crossover exchanges partial service selections while mutation can change a service for a randomly chosen task with a certain probability.
(2) A divide-and-conquer approach [29]. This is performed by grouping workflow nodes into partitions, distributing the deadline and using local solutions for the partitions. It was compared [29] to a greedy algorithm such that all tasks have the same deadline and a deadline algorithm in which deadlines for tasks on the same level are the same. It was shown that the proposed algorithm offers similar execution time and returns cheaper solutions for pipeline, parallel, and hybrid applications compared to the others. Abrishami et al. [2] present a workflow scheduling algorithm that is executed in two phases: deadline distribution and planning. The first step distributes the defined deadline among tasks so that each one finishes before its own. Then cheapest services are selected to meet particular deadlines.
(3) Integer linear programming ILP [3, 4, 31] can be used to solve the problem where integer variables denote which service has been selected for execution of a particular task. However, it imposes linearity constraints in the model. Canfora et al. [8] suggest that genetic algorithms are preferred for a large number of concrete services per abstract service, otherwise integer programming is better.
(4) Finding a service mapping and improving it iteratively. Sakellariou et al. [20] propose LOSS and GAIN that first obtain a solution that results from optimization of only time or only cost to meet the cost constraint and optimize the execution time. This is then improved iteratively until a valid solution is found. Similarly, Kyriazis et al. [16] propose a mapping of workflow processes to service instances so that the user's requirements are met concerning availability level or cost. Subsequently, improvements of the first solution with a better level of QoS replace the previous solution.

2.2 Existing work on storage-aware workflow management

Workflow management systems do offer various ways of handling data. For instance, as indicated in [11] Gridbus accepts input data in the form of values, files, or data streams [18]. Input data is processed as it arrives [1] implementing data streaming. Data patterns such as one-to-one or all-to-all were introduced to workflow management systems to handle data coming from multiple sources [14]. However, irrespective of these various types and patterns that define how data can be partitioned among parallel paths and then integrated, various constraints on data processing should also be considered.

Singh et al. [21] present solutions for optimization of data usage when executing data-intensive workflow applications in distributed systems. It is assumed that particular files are located on and transferred between resources from which can be used for workflow jobs/tasks. The authors propose a technique to remove data files during workflow execution when the files are no longer needed to decrease the data footprint

of the workflow. However, a large number of clean-up jobs, even larger than the number of workflow tasks may be needed. Additionally, work [19] presents an algorithm that maps workflow tasks onto resources with enough space to minimize the workflow execution time. The algorithm prioritizes resources capable of storing data for a task in terms of execution times and uses the aforementioned clean-up algorithm.

The work in [19] was analyzed and further extended in [5] where two problems are considered. The first one is the Minimum Makespan with Storage Constraints problem, which minimizes the workflow execution time and assures that the total storage used by the workflow does not exceed the limit. The order of clean-up jobs is considered with the total size of files used by ancestors of the clean-up job. If it exceeds the storage limit, the algorithm needs to wait for release of pending clean-up jobs. Additionally, two versions were considered: one with explicit ordering of computational jobs—ancestors of clean-up jobs and another one with no ordering of these. Bharathi et al. [5] also consider minimization of a weighted metric of the product of workflow execution time with the maximum number of processors used and the product of workflow execution time with the maximum storage used during execution. Genetic based algorithms were proposed to solve these problems.

Yuan et al. [30] consider the problem of storing intermediate data used within a workflow from the cost point of view. Storage of intermediate data involves a cost. On the other hand, after data has been deleted to save the cost, a cost of regeneration of this data will be involved if it is needed again. The paper presents a dependency based intermediate data storage strategy, and shows that it results in a smaller total storage cost than strategies: store all, store none, store often used, and store high generation cost data chunks.

2.3 Motivations for a new extended model

This paper extends the aforementioned works by taking into account storage constraints of not only the resources from which services access data and the workflow data footprint, but also storage constraints of the workflow execution engine and data caching when data is processed as streams.

We assume that services executed in workflow nodes are provided by providers from their own resources such as servers or clusters on which the services are installed. The goal is to optimize QoS that involves the execution time and cost. In this case, the resources may have limits for storage of data chunks processed by the services. In some cases, if more processors/cores are available, several service instances may be executed in parallel, but must still observe the storage limitations. In this case, the storage limits affect the workflow execution time and higher limits would require a higher cost per service. This is incorporated into the scheduling algorithm that follows individual limits of resources. Thus, it is also straightforward to monitor and maintain the total workflow data footprint below a given threshold, if needed.

The contribution of this work is as follows:

(1) A workflow model that allows definition of data streaming on a per task basis and a workflow execution engine in the BeesyCluster middleware [11] that supports it.

(2) Incorporation of several storage limits into the previously developed workflow execution engine in BeesyCluster [10]:

- configurable storage limit of the workflow execution engine,
- storage constraints for the resources on which services are installed; this also allows monitoring and limiting the workflow data footprint at runtime.

Previously, the author codeveloped two solutions: a centralized execution engine in BeesyCluster based on the Java EE technology [10] and a distributed JADE-based engine managed by a group of distributed software agents in BeesyBees [12]. In these cases, a Java EE server or JADE containers have storage limits. Higher limits will involve more cost added to the cost of the workflow.

(3) Implementation of a caching technique that allows execution of next data chunks by a preceding service if the following service is not yet ready to accept output data of the preceding service due to storage limitations. This is solved by caching output results of a preceding service until the next service can accept it. This speeds up processing if data is processed in streams as successive data chunks.

(4) Allocation of the budget for the workflow not only to the services as in the already known approaches, but also to the storage of the workflow execution engine that also acts as cache space. A method is proposed how to distribute the budget so that the workflow execution time is minimized.

3 Proposed model

Following the aforementioned workflow application model used in the literature [6, 24, 25, 27, 29] and the previous works of the author [10, 11], we now focus on how the execution engine executes services and transfers data between services when data is coming in a stream, i.e., as successive data chunks. As shown in Fig. 1, a workflow execution engine executes workflow tasks in the order determined by the edges of a directed acyclic graph that models the workflow. For each task t_i, one service s_{ij} is selected for execution out of possibly several functionally equivalent services in set S_i. Each service has its own QoS parameters such as cost c_{ij} and execution time t_{ij}. It accepts output data from its predecessor(s) and transfers its own output data to following services. Each service s_{ij} processes data of size d_{ij}. Globally, for each task such service is selected so that a certain QoS goal is optimized with possibly additional QoS constraints. For example, the workflow execution time can be minimized with a bound on the cost of selected services [10, 11].

In this work, rather than on service selection, we focus on transfer of data between services such as s_{ij} and s_{kl} selected to execute tasks t_i and t_k respectively, especially on storage limits and storage costs of both the resources the services use and the workflow execution engine.

The following extensions are proposed for flexible and efficient management of data transfer considering constraints on the capacity of locations where the services are installed and constraints of the execution engine:

(1) *Data streaming per task*: can be enabled or disabled on a per task basis. Data streaming denotes that processing of particular portions of input data can be performed independently as soon as the given portion of data is available. It is then

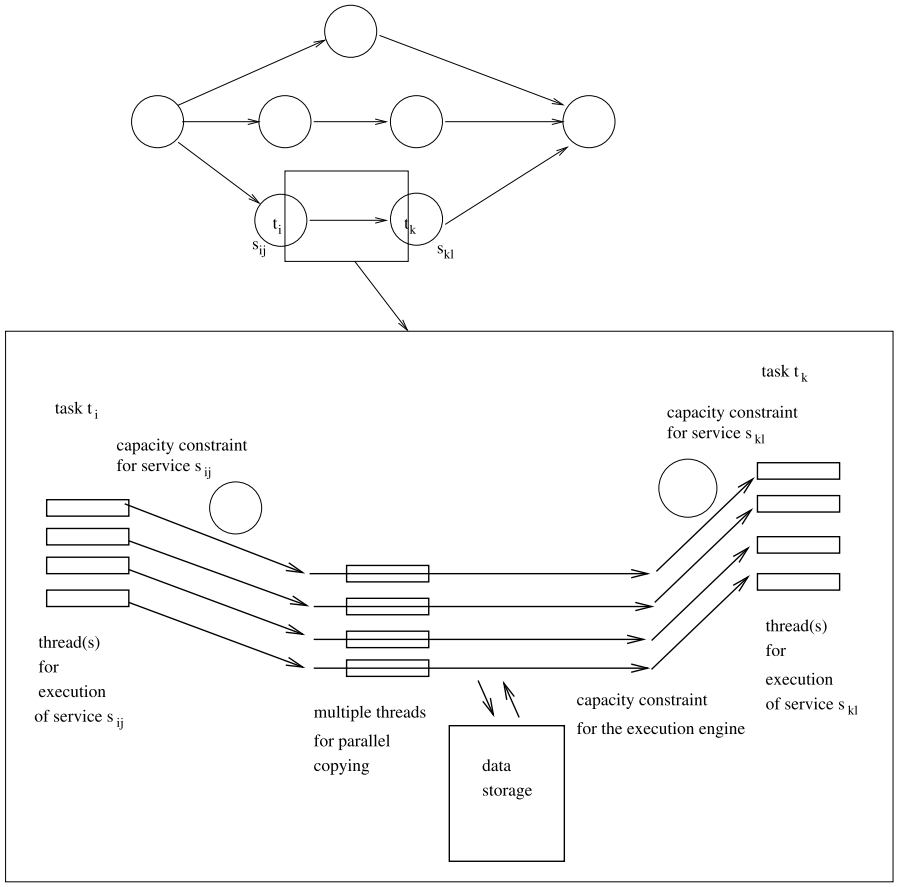

Fig. 1 Copying of data to a service with data streaming enabled

forwarded to a following service(s) promptly after processing. This is done independently from processing other data chunks by separate threads. As an example:

- if the task is to convert input images from TIF to JPG, then it can be processed in the streaming mode. As soon as there is an input file or input files, the service for the task can convert them and send to successive task(s),
- if the task is to produce a web album out of the input images, then it must be performed in the non-streaming mode. All input images must be ready in order to produce this album.

Note that one workflow application can contain tasks both with streaming and without streaming modes.

(2) *Storage constraints for both services and workflow execution engine*:
   (a) Maximum data size $ds_{ij}^{\max}(t)$ that can be stored and processed at the same time by service s_{ij}. This is related to the service and the capacity of the

server on which the service was installed. The cost of this storage is included in the cost of the service.

(b) The workflow data footprint at moment t: $\sum_{i,j} ds_{ij}^{\text{cur}}(t)$ can be maintained below a given threshold where $ds_{ij}^{\text{cur}}(t)$ denotes the size of data processed by service s_{ij} at moment t.

(c) Maximum data size $\text{WEC}(t)$ that can be stored by the workflow execution engine in the intermediate storage at time t.

(3) *The size of data packet* $\text{SDP}_{i;k}$ sent at once in a data stream between tasks t_i and t_k; after sending of each data packet, processing of this packet by the following service is initiated. The size of the data packet is measured in the number of chunks/files in the packet.

(4) *The number of parallel data streams* PARDS for data copying from:

- an initial data location dl to the service selected for the given first task t_i of the workflow application – $\text{PARDS}_{dl;i}$,
- between services selected for execution of subsequent tasks t_i and t_k— $\text{PARDS}_{i;k}$.

This parameter is related to the number of data portions being sent in one stream by a thread assigned to the particular data stream. This number of data portions is determined by the number of data portions ready to be sent to a particular location divided by the requested number of parallel data streams.

For each service, there are threads processing data packets as they arrive in the streaming mode. The total data size of data processed by these threads at the moment must not exceed $ds_{ij}^{\max}(t)$ and the total data size used by the workflow $\sum_{i,j} ds_{ij}^{\text{cur}}(t)$ must not exceed the maximum size for the workflow data footprint if specified. Output data of size d produced by a thread responsible for processing of a data chunk is handled by another thread which is responsible for copying the data to the following service s_{kl}. The thread can copy the data only if $ds_{kl}^{\text{cur}} + d \leq ds_{kl}^{\max}$. Otherwise, two solutions are possible:

- The data can wait in the location of the previous service until there is sufficient data space in the storage of a subsequent service.
- The thread can copy the data to the intermediate storage provided that $\text{weccur}(t) + d \leq \text{WEC}(t)$, $\text{weccur}(t)$ denotes the current data size in the intermediate storage. This procedure decreases the size of data handled by the preceding service s_{ij}, and consequently allows copying new data chunks to it and initiating processing as soon as possible.

Considering the proposed extensions, the formal scheduling model can now be presented as follows (with the summary of notation shown in Table 1):

input data:

a directed graph $G(T, E)$ that models a workflow application where vertexes T correspond to tasks (denoted by t_i) and edges to dependencies between the tasks. Each task must have one out of one of the following working modes assigned to it:

regular—means that results from the task will be sent to following task(s) only after all input packets have been processed,

Table 1 Summary of notation

Symbol	Explanation
t_i	task i of the workflow application $_{1\le i\le \lvert V\rvert}$
$S_i = \{s_{i1}\cdots s_{i\lvert S_i\rvert}\}$	a set of alternative services each of which is capable of execution of task t_i, out of which only *one* must be selected to execute task t_i $_{1\le i\le \lvert V\rvert}$
$c_{ij} \in R$	cost of processing a unit of data by service s_{ij} $^{1\le j\le \lvert S_i\rvert}_{1\le i\le \lvert V\rvert}$
P_{ij}	provider of service s_{ij} $^{1\le j\le \lvert S_i\rvert}_{1\le i\le \lvert V\rvert}$
N_{ij}	node on which service s_{ij} runs $^{1\le j\le \lvert S_i\rvert}_{1\le i\le \lvert V\rvert}$; each s_{ij} is installed on a computing node—N_{ij}
$ds_{ij}^{\max}$	storage capacity of the host/cluster on which the service has been installed
$sp_n \in R$	speed of node n
d_{input}	size of the input data to the workflow
d_i	size of data required and processed by task t_i $_{1\le i\le \lvert V\rvert}$
d_{ij}	the size of data processed by service s_{ij}
WEC(t)	maximum data size that can be stored at time t in the storage used by the workflow execution engine
$\text{SDP}_{i;k}$	the size of data packet sent at once in a data stream between tasks t_i and t_k
$\text{PARDS}_{dl;i}$	the number of parallel data streams for data copying from an initial data location dl to the service selected for the given first task t_i of the workflow application
$\text{PARDS}_{i;k}$	the number of parallel data streams for data copying between services selected for execution of subsequent tasks t_i and t_k
$t_{\text{workflow}} \in R$	time when the last service finishes processing the last chunk of data
$B \in R$	available budget
$B_{\text{WEC}} \in R$	budget for the storage size of the workflow execution system taken out of the initial budget B

streaming—means that input data packets are processed within the task immediately and forwarded to following task(s) as soon as possible,

sets of services S_i assigned to workflow tasks. Each set S_i contains a set of services s_{ij} out of which one needs to be selected for execution of task t_i. Each service has the following attributes assigned to it:

execution time t_{ij},

cost c_{ij},

storage capacity $ds_{ij}^{\max}$ of the host/cluster on which the service has been installed,

possibly other QoS metrics that can be incorporated into the QoS scheduling goal,

input data of size d_{input},

maximum data size WEC(t) that can be stored at time t in the storage used by the workflow execution engine,

budget B that can be spent on services and storage,

budget B_{WEC} taken out of the initial budget B for the storage size of the workflow execution system,

output data (such that all required constraints are met):

assignment of a service s_{ij} **to each task** t_i,

processing of each data packet by the selected service at a particular point in time,
copying of each data packet to a following service at a particular point in time,
setting optimal parameters SDP_* (size of data packet sent between tasks) **and** PARDS_* (the number of parallel data streams),
criteria: minimization of the workflow execution time t_{workflow} **while keeping the total cost of services below budget** B **i.e.** $\sum c_{ij} d_{ij} \leq B$.

The previous works by the author [10, 11] present how services should be chosen to optimize QoS goals. Note that the input data of a predefined size d_{input} is considered. Data files are packed into data packets and forwarded through the workflow along a path selected by the scheduling algorithm. In this case, we consider minimization of the workflow execution (until the last data packet has been processed by the last service) time with a bound on the total cost of selected services. We have budget B which can be spent on running the workflow. Please note that usually the workflow scheduling problem considers that particular workflow tasks process data of certain size. The author previously introduced an extension of this model [11] in which the algorithm can determine how data chunks are distributed among parallel data paths i.e. the algorithm obtains what final data sizes each task will process to obtain the shorted possible execution time. This is also applicable in this work. However, this paper extends the approaches mentioned in Sect. 2 and also the algorithm developed by the author in [11] by allocation of a part B_{WEC} of the initial budget B for the storage size of the workflow execution system. Allocation a part of the cost to larger storage of the execution engine that acts as cache can speed up execution of the workflow. Then the remaining part $B - B_{\mathrm{WEC}}$ can be spent on the services. Let $\mathrm{wet}_{B_{\mathrm{WEC}}}(B - B_{\mathrm{WEC}})$ denote the workflow execution time in such a scenario such that the total cost of selected services for processing data is below the given budget, i.e., $\sum_{i,j} c_{ij} d_{ij} \leq B - B_{\mathrm{WEC}}$. The goal is to minimize the workflow execution time which can be accomplished by checking possible allocations of budget B to B_{WEC} and services for the smallest value of

$$\min \mathrm{wet}_{B_{\mathrm{WEC}}}(B - B_{\mathrm{WEC}}). \tag{1}$$

For each B_{WEC}, $\mathrm{wet}_{B_{\mathrm{WEC}}}(B - B_{\mathrm{WEC}})$ needs to be computed by a QoS service selection algorithm such as the one proposed by the author in [11]. This algorithm also partitions data among parallel workflow paths allocating d_{ij}'s to particular services to minimize the workflow execution time. In this case, though, it needs to compute the workflow execution time for the services considering the storage constraints and caching as presented next.

4 Design and implementation of a multithreaded execution engine with data streaming, caching, and storage constraints

4.1 Existing workflow management and optimization in BeesyCluster

From the implementation point of view, the proposed solution extends the workflow management system in BeesyCluster, implemented by the author before [10]

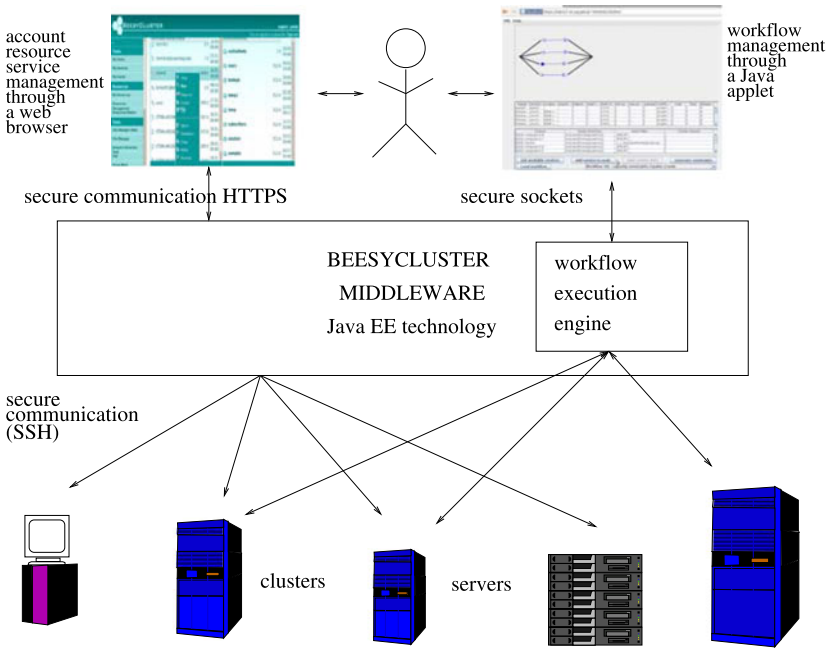

Fig. 2 Existing system architecture

and shown in Fig. 2. In general, BeesyCluster acts as a middleware that allows users to access and use distributed clusters and servers on which they have system accounts. Access is possible via WWW or web services. Each user can manage system accounts to which they have access such as edit files, manage directories, compile, and run applications (either with a text or graphical interface). Applications from clusters and servers can be published as BeesyCluster services and assigned QoS parameters such as execution time, cost etc. Providers can assign privileges to other BeesyCluster users on these QoS terms. This is depicted by the left side of Fig. 2.

Furthermore, as depicted by the right flow in Fig. 2, BeesyCluster contains a workflow management system that allows: definition of workflows according to the aforementioned model, assignment of services capable of execution of particular workflow tasks out of own or services made available by others, definition of QoS goals, automatic selection of services, and workflow execution in a distributed environment with dynamic reselection if services have become unavailable. Two workflow execution engines were developed: one implemented in the Java EE technology [10] (embedded into the middleware and shown in Fig. 2) and one in which distributed software agents manage workflow execution [12].

However, the solution lacked support for flexible data streaming in a workflow and a variety of options and constraints for data storage which is proposed and introduced in this work according to the model presented in Sect. 3.

4.2 Proposed solution

The implementation of the workflow management module extends the solution proposed by the author in [10].

From a bird's eye view, the new workflow management subsystem consists of the following components:

(1) Servlets allowing: management of already defined workflow applications, browsing statuses, and results of already started workflow instances.
(2) A client application implemented as a Java applet that allows to create, edit, and save workflow applications on the server side. Services available to the user are downloaded from BeesyCluster and can be assigned to workflow tasks in the editor.
(3) A new multithreaded workflow execution engine.

The BeesyCluster architecture allows publishing of services from both high performance clusters with larger storage spaces (usually in GB or TB) but also from typical servers or even PCs used by other users or their owners at the same time. This is much like donation of a part of own resources to the BOINC platform in volunteer computing. In the latter case, the user-provider may want to restrict used storage space to even MBs. It is worth to note that the proposed solution supports setting various limits for various services thus considering both types of resources in one workflow.

Execution of a workflow is handled by threads each of which is responsible for running one workflow node/task for either all its input data (non-streaming mode) or a part of it (streaming mode). In the latter case, there can be several threads per one workflow task handling parts of the data assigned to the task. Each thread is designed to handle one task of the workflow application. Activation of a task is preceded by sending some input data files to a designated directory on the cluster/server on which the service is installed. The thread is responsible for processing the packets using the service and forwarding to following workflow tasks. A task invokes a new task i.e. another thread after it has finished processing its portion of data. Each thread launches remote services over SSH using the `jsch` library. When a thread shown in Fig. 3 needs to send its full output data (as in the nonstreaming mode) or a part of output files that are already available to another task, a `DataCopier` object is created, to which these output files are passed. At the same time, a separate thread is launched to finalize the process of sending this data to following nodes (Fig. 4). The `DataCopier` thread does the following:

(1) Output data from the previous service is transferred to the BeesyCluster cache which frees the storage of this service and allows processing of new chunks of data irrespective of the following services.
(2) As soon as the storage capacity used on the node on which the following service is running is lower than its limit, data is copied there and a new thread is invoked for processing this data. Note that a new thread is spawned for copying data. If the storage limit of the following service has

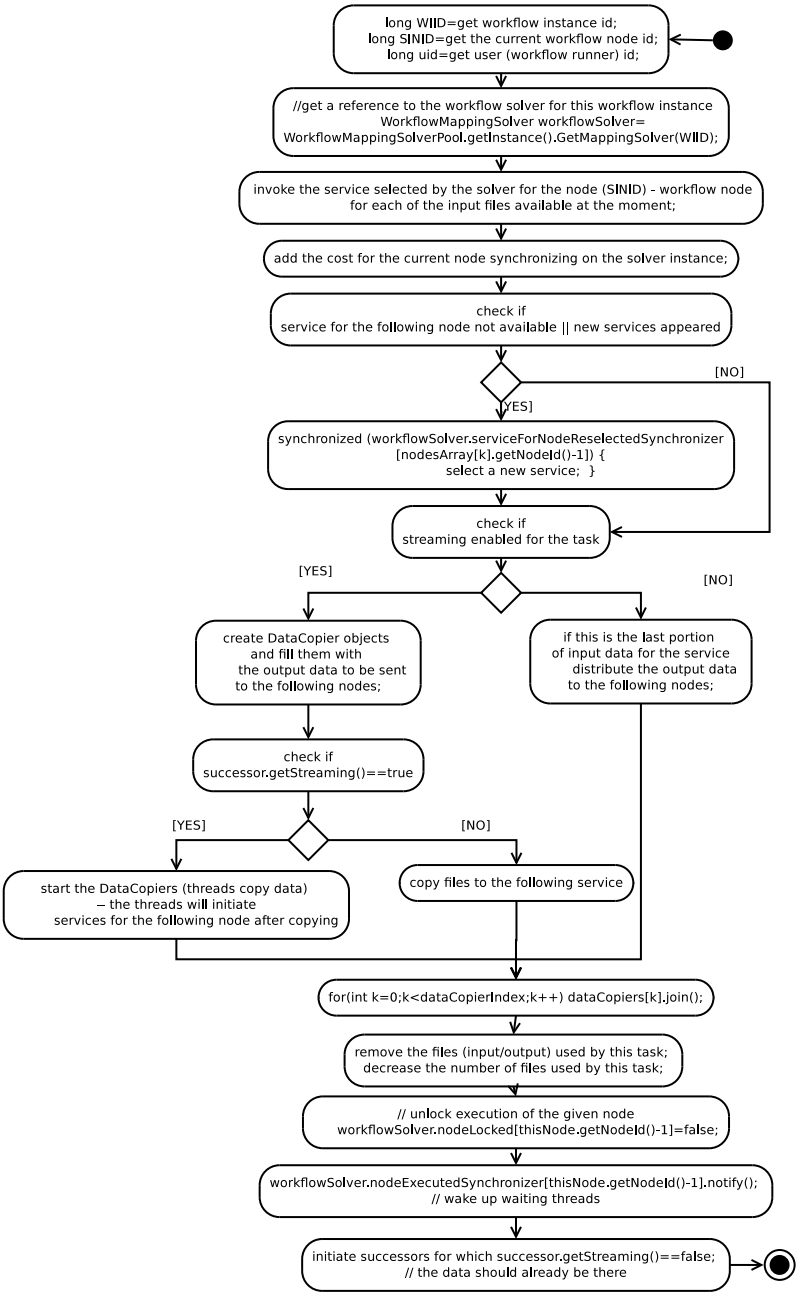

Fig. 3 Activity diagram for the thread responsible for execution of workflow task

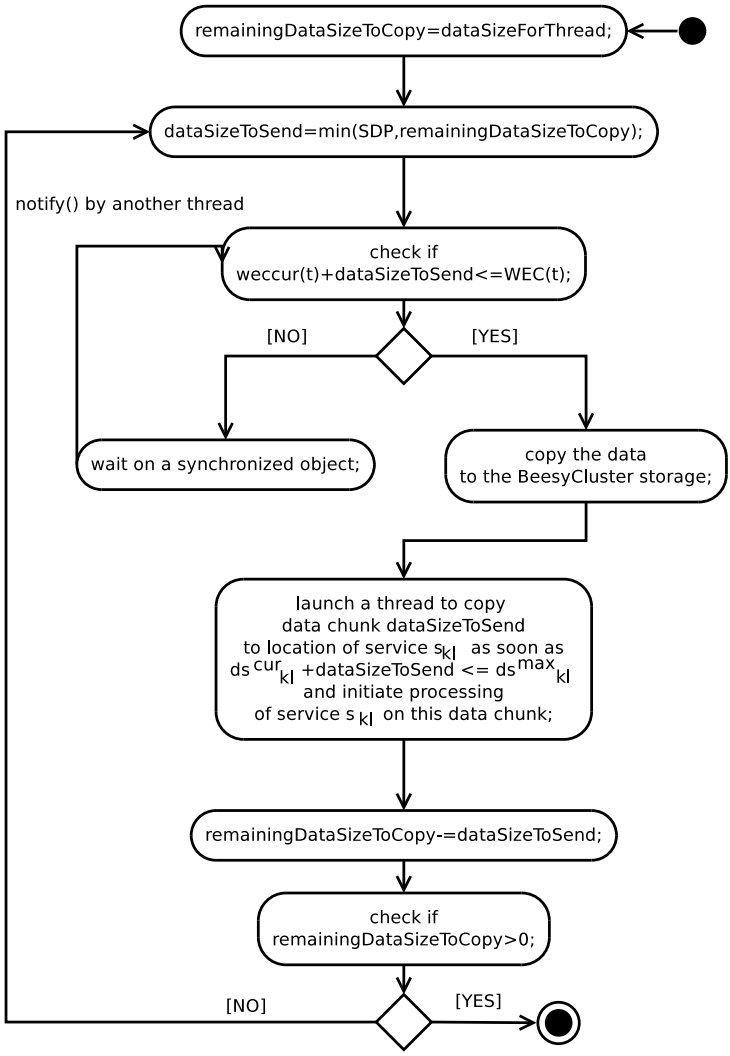

Fig. 4 Activity diagram for a data copier thread for copying data between services with storage constraints and data caching

been reached, the `DataCopier` needs to wait. It is then woken up by the notify() method as soon as some data has been processed and copied to following nodes.

(3) The maximum size of the data chunk sent to the following service is also observed and followed by the `DataCopier`. It makes sure that the data size does not exceed the value of `maximumDataSizePerCopy` ($SDP_{i;k}$).

The streaming mode was incorporated into the code responsible for execution of a single service as shown in Fig. 3. Namely, as soon as new input data chunks are ready

for processing by a particular workflow node, a `DataCopier` copies this data and invokes a new task observing the imposed storage limits.

A certain number of parallel data streams for copying data between services is implemented as launching multiple instances of `DataCopiers` that need to synchronize on a shared object for the following workflow node to make sure that the total size of data sent to this node does not exceed its storage limit $ds_{ij}^{\max}$.

All the steps of workflow execution are monitored and are recorded in the BeesyCluster database. This allows measurements of all communications of data between services along with execution times of particular services. A servlet for management of instances of started workflow applications displays statuses and current accumulated execution times (since started) of particular instances.

4.3 Storage-aware QoS service selection algorithm

The workflow execution engine must select a service to each workflow task such that the optimization criterion is optimized. In this case, we consider minimization of the workflow execution time with a bound on the total cost of running the workflow. The scheduling algorithm should consider storage constraints as these affect the workflow execution time.

For this purpose, the genetic algorithm approach previously proposed by the author in [11] was modified for optimization of service selection in this work. The new contribution compared to the previous version is consideration of the storage constraints in the evaluation of a chromosome, i.e., in computing the workflow execution time. As mentioned above, data streaming, storage constraints and caching affect the execution time of the workflow and the cost if intermediate storage space is used. The execution time of the workflow can be calculated as follows. First, let us introduce the following variables:

- $N(t_i)$—a set of tasks which follow task t_i in graph G,
- V_{start}—a set of initial workflow tasks,
- TH_i—a set of threads currently copying data to or executing task t_i,
- th_{ij}—jth thread responsible for handling a chunk of data for task t_i, either copying the data chunk to the service chosen for task t_i or processing the data.

The algorithm for computing the workflow execution time for the given services for workflow tasks and storage constraints can be formulated as shown in Fig. 5 simulating data flows, similarly to pushing data flows in maximum flow problems [22].

Note that the genetic based algorithm works as proposed in [11]. A population of chromosomes is generated each of which represents assignment of a service to each task. Generations of chromosomes are created one by one with crossover and mutation. The fitness function aims at minimization of the workflow execution time with the total cost of selected services below $B - B_{\text{WEC}}$.

In general, performance of communication depends on the network topology, latencies and bandwidth of links between locations on which run services chosen for execution of particular tasks. The scheduling algorithm considers (Fig. 5) these parameters in computing the final workflow execution time for a particular set of selected services and storage constraints. Depending on the environment chosen, e.g.,

```
currenttime=0;
for each task t_i in V_start {
  create threads th_ij s for copying data to service s_ih
  chosen to execute t_i such that the total data size does not exceed ds_ih^max;
  add each thread th_ij to TH_i;
}
while(at least one TH_i not empty) {
  tmin=Double.MAX_VALUE;
  for each task t_i such that TH_i not empty {
    for each thread th_ij in TH_i {
      // check which th_ij finishes first (this
      // includes both copying output data to the task and execution)
      if (th_ij is execution of task t_i) {
        time_ij=execution time of th_ij on the data assigned to th_ij;
      } else { // th_ij is copying data to task t_i)
        time_ij=data copying time
        to the location of service s_ih chosen to execute t_i --
        send maximum SDP and do not exceed ds_ih^max;
      }
      if (tmin>time_ij) {
        tmin=time_ij; firstthread=th_ij;
      }
      // tmin is the next time step in the execution
    }
  }
  currenttime+=tmin;
  remove firstthread from its TH_i;
  if (firstthread was executing (not copying) data for task t_i) {
    for each t_k ∈ N(t_i) {
      create threads th_kl s for copying data to service s_km
      chosen to execute t_k
      such that the total data size does not exceed ds_km^max;
      add each thread th_kl to TH_k;
    }
  } else { // firstthread was copying data to task t_i
    add a thread th_kl to TH_k that is responsible
    for execution of service s_km chosen to execute t_k
    for the data just copied;
  }
}
workflowexecutiontime=currenttime;
```

Fig. 5 Evaluation of the workflow execution time

services installed on cluster nodes or clusters located in one HPC center or services on commodity servers or PCs distributed geographically across continents, this may greatly impact performance. It is then up to the scheduling algorithm to select the best possible configuration through optimization and considering the performance model of the links between nodes.

5 Experiments

Section 5.1 specifies the environment used for subsequent tests. Section 5.2 describes testing of the proposed mechanisms regarding constraints on storage size and caching. Then Sect. 5.3 presents test cases for a real world distributed workflow for parallel processing of digital images including impact of the data packet size in data

streaming on the execution time and determination of good budget distribution between intermediate storage and services to minimize the total workflow execution time.

Firstly, each workflow tested was drawn and assigned particular services in a workflow editor in BeesyCluster. This is depicted in Figs. 6, 8, 10, and 12. It must be noted that initial data is indicated by pointing a directory on one or more clusters available to the user. To make it clearer, it has been marked that data from the original location (one in the tests) was partitioned among the parallel paths. Furthermore, results from the paths are copied to one designated location which has been reflected in the figures (normally not shown in the editor).

Then workflow scheduling in BeesyCluster was launched which initiated the scheduling algorithm and subsequent workflow execution. Data from the initial space is then partitioned and sent to initial nodes drawn in the editor. Similarly, data from the last nodes indicated in the workflow in the graph drawn in the editor is copied to one designated output directory from which it can be fetched by the user using the file manager module available in BeesyCluster (shown in Fig. 2).

5.1 Testbed environment

All the tests were performed on clusters that were formed out of parts of a department cluster. Each node of this cluster features 2 dual core Intel Xeon CPUs at 2.8 GHz with Hyper Threading, 4 GBs of RAM running CentOS release 5.4 (Final), 64-bit, Linux 2.6.18–164.6.1.el5. The services assigned to connected workflow nodes were located on distinct clusters. The network uses Gigabit Ethernet. Each service is installed on a dedicated computing node. Depending on the configuration, as mentioned in the descriptions of the experiments, instances of a service processing data chunks may use either one or more cores of the node.

5.2 Testing of the proposed units

5.2.1 Impact of BeesyCluster storage size on the workflow execution time

Since BeesyCluster acts as a proxy in data transfers between distributed services, output data from a service needs to be copied to the location where the following service is installed. If the workflow is complex and potentially data of large sizes needs to be copied between many pairs of services at the same time, the storage capacity of the BeesyCluster management layer will have an impact on the total execution time, especially if there are many parallel paths in the workflow.

Figure 7 presents workflow execution times for various storage limits for intermediate storage and for the workflow application depicted in Fig. 6. It should be regarded as a communication pattern that appears in many practical workflow applications. The parameters of the experiment are as follows. 80 input files (data chunks) of either 12 MB each or 100 MB each were generated (the size is important, not the content) for which two graphs are shown. Additionally $ds_{ij}^{\max} = 10$ files, PARDS $= 8$, SDP $= 1$ file, streaming enabled, instances of each service processing various chunks of data run on one processor core. Each service had the same processing time per

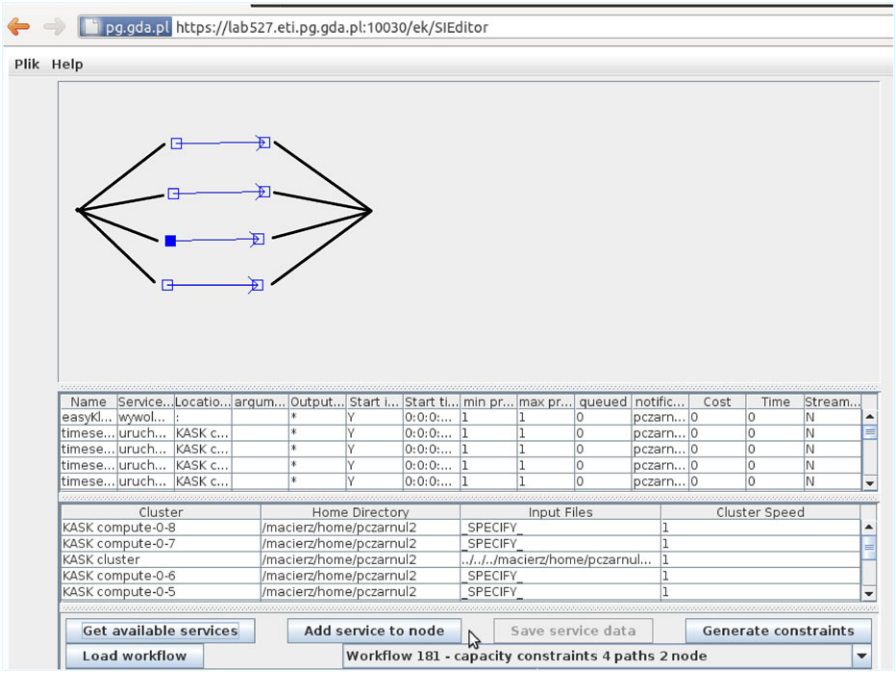

Fig. 6 Testbed workflow communication pattern

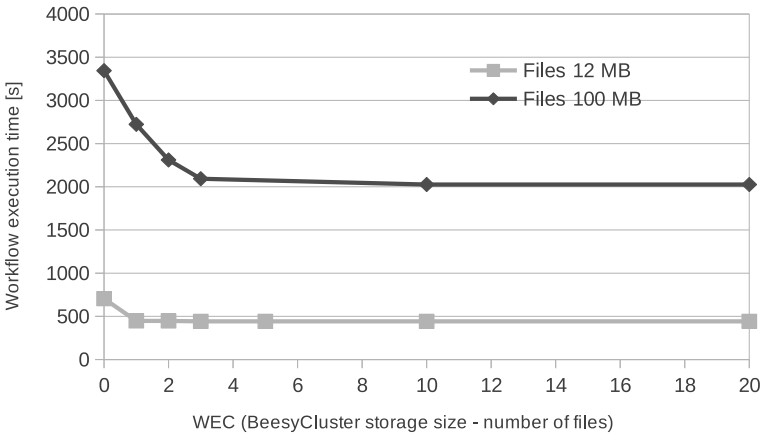

Fig. 7 Workflow execution time (s) vs. BeesyCluster storage size

data chunk in this example. The goal was to minimize the workflow execution time, same costs were considered for services with an infinite bound (budget). It can be seen very easily that offering a larger storage capacity of the BeesyCluster execution engine allows to decrease the workflow execution time.

5.2.2 *Impact of multithreaded copying, caching, and storage constraints on the workflow execution time*

If output data from a service is ready but cannot be copied to a subsequent service yet, e.g., due to the limitation of the data size that can be processed in the latter at the same time, processing of the first service may be put on hold for the same reason. To solve this issue, the `DataCopier` presented in Sect. 4 copies such data to the BeesyCluster storage and removes from the location of the service assigned to the previous node. This allows processing of new data packets by this service as soon as possible since more data space is now available. For each data packet (the size is configurable), a new thread is launched that copies the data to a following service when the latter is ready to accept new data, i.e., when its current data size and the size of the new data does not exceed its storage capacity. As soon as this is the case, the thread copies the data to the location of the following service and launches its execution. If there are many threads waiting for availability of the storage in the following service, they are synchronized.

The parameters of the test are as follows: $ds_{ij}^{\max} = 2$ files, PARDS $= 2$, SDP $= 1$ file, WEC $= \infty$, streaming enabled, multiple instances of a service working on chunks of data can work on multiple cores of processors in the node. The test was performed for the workflow application presented in Fig. 8 and 80 input files of 12 MB each. As in the previous case, it should be treated as a common pattern in workflow

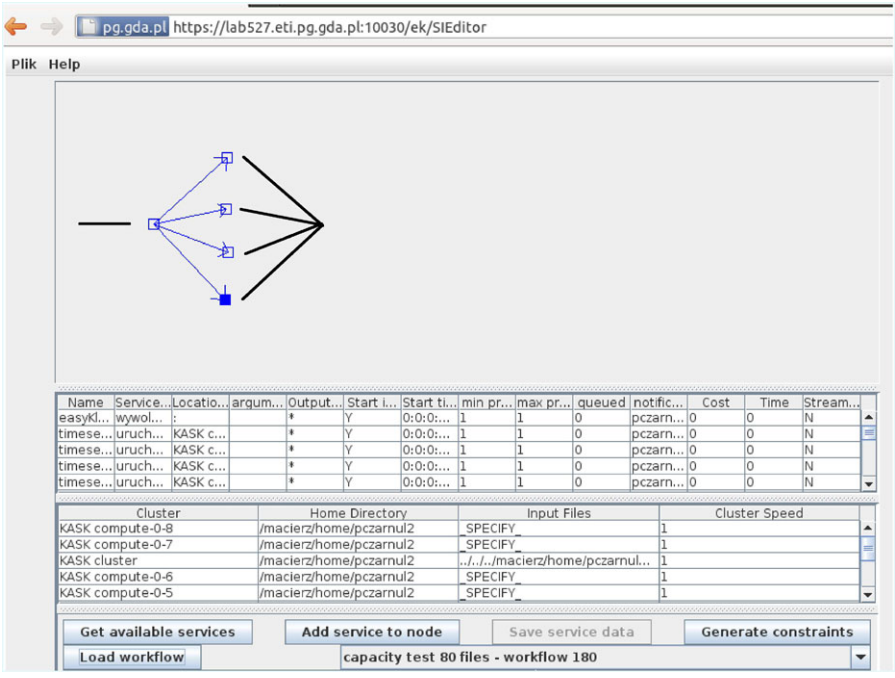

Fig. 8 Testbed workflow communication pattern for workflow execution time (s) vs. maximum size of data in node experiment

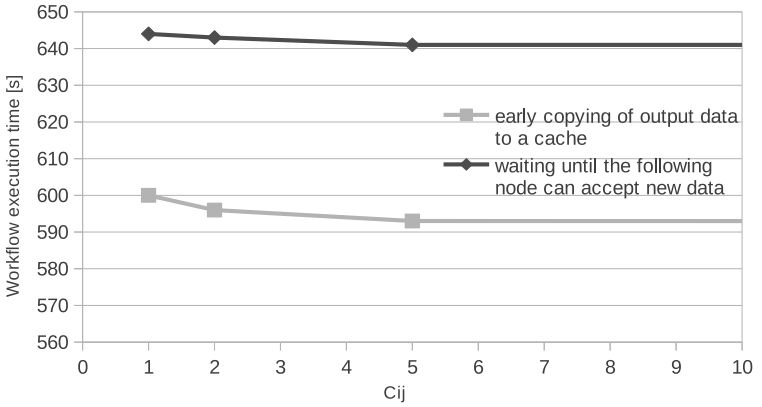

Fig. 9 Workflow execution time (s) vs. maximum size of data in node

applications. The goal was to minimize the workflow execution time, same costs were considered for services with an infinite bound (budget). Results are presented in Fig. 9. Apparently, the caching technique allows to reduce the workflow execution time considerably for all storage constraints of particular services $ds_{ij}^{\max}$. Additionally, as could be expected, the execution time decreases with the increase of storage limits for the services $ds_{ij}^{\max}$. A higher limit allows to start copying of data to following services earlier.

5.3 Experiments for a digital photography workflow with streaming and storage constraints

Further tests, including testing the impact of the packet data size on the execution time and a comprehensive test for determination of cost allocation to services and storages, were performed for a real workflow for parallel processing of digital images. Digital images in RAW formats produced by digital cameras are usually processed in a pipeline with at least a few steps, which were implemented as services invoked one after another in a sequence:

(1) Conversion of a RAW image to a 16-bit TIFF implemented as script `dcraw -T $1` where `$1` denotes an input file and `dcraw` is a program for RAW image conversion,
(2) Normalization of the TIFF image implemented as script `convert $1-normalize $1.TIFF` where `convert` is ImageMagick's command line tool.
(3) Resizing, sharpening, and conversion to a final JPG file implemented as script `convert $1 -resize 600x400 -sharpen 1x1.2 -quality 97 % $1.jpg`.

For further experiments in this section, workflow applications with several parallel paths including these steps were defined with services installed on various clusters. Multiple instances of a service working on chunks of data can work on multiple cores of processors in the node.

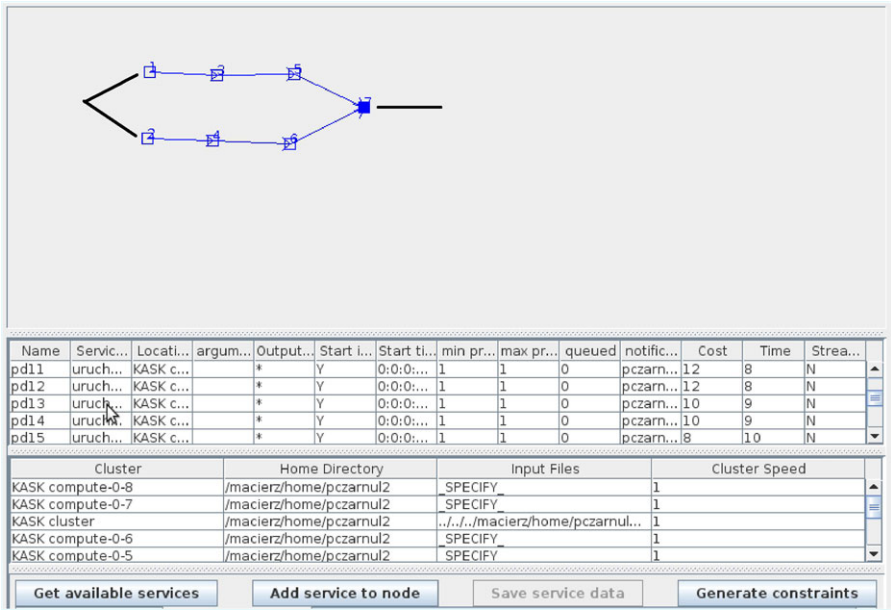

Fig. 10 Testbed workflow for digital photography tests

5.3.1 Size of data packet

In this experiment, the goal was to measure the impact of the size of a data packet in the streaming mode. The workflow application from Fig. 10 was used with 2 parallel paths, $ds_{ij}^{\max} = 100$ files, WEC $= \infty$, streaming enabled, 80 input files of 12–13 MB each. In this case, the images were real photographs taken by a modern DSLR and stored in a RAW format. The goal was to minimize the workflow execution time, and the same costs were considered for services with an infinite bound (budget). The final node was added to integrate output files into the user specified location. Figure 11 presents obtained results. Smaller data packets allow faster initiation of successive tasks and services and thus shorter execution times with the exception of very small data packets with one file only.

5.3.2 Impact of storage sizes on workflow execution time

Further experiments are meant to investigate if and how different distributions of the available budget B among services and storage affect the total workflow execution time. Namely, allocation of more budget to services and less to intermediate storage may result in faster processing and slower communication and vice versa. The goal is to investigate whether different relative costs of services and storage require different distribution of budget B.

For the experiments in this section, the workflow application depicted in Fig. 12 was used. There are 9 parallel paths each of which applies the three steps of digital photo processing. However, the workflow application is not balanced as the paths had

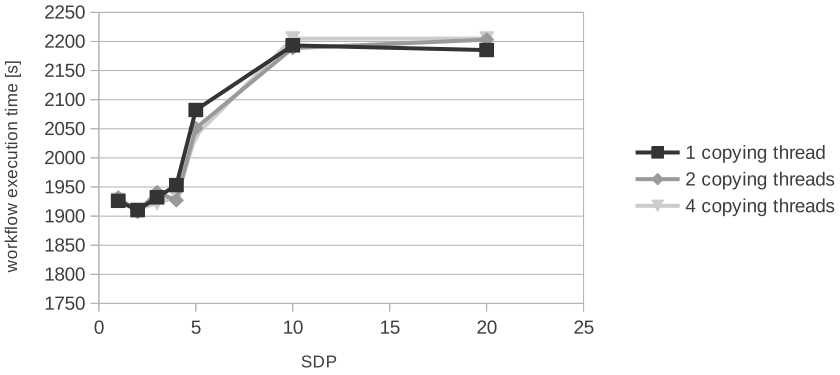

Fig. 11 Impact of data packet size on the workflow execution time

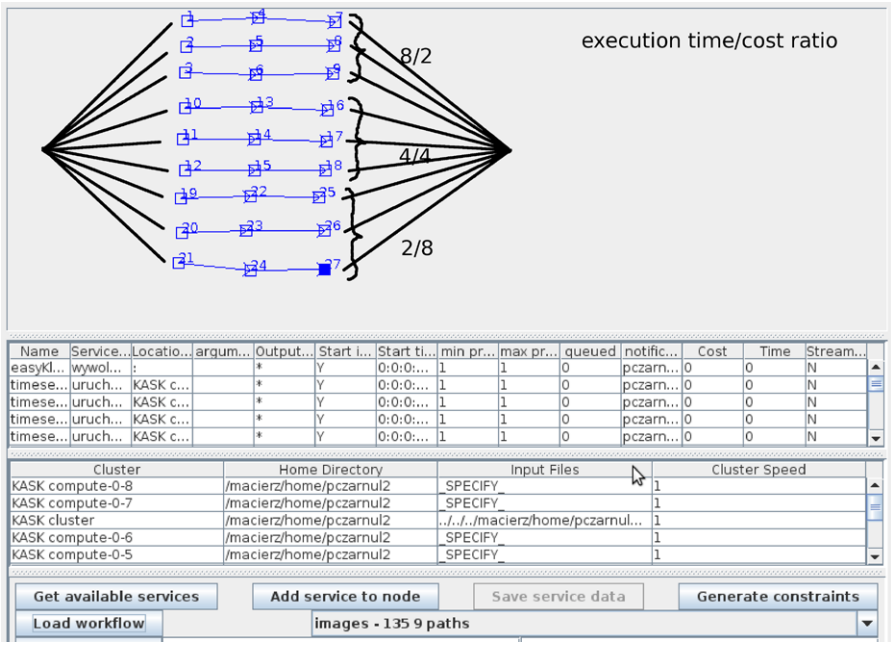

Fig. 12 Testbed workflow for digital photography tests

services with various execution time/cost ratios assigned to them as follows: 3 paths with 8/2, 3 paths with 4/4 and 3 paths with 2/8. 135 input files of 12–13 MB each were used as input for a total of 1.7 GB of data. The goal was to minimize the workflow execution time while keeping the total cost of selected services for processing data below the given budget, i.e., $\sum_{i,j} c_{ij} d_{ij} \leq B - B_{\mathrm{WEC}}$. d_{ij} corresponds to the number of data files processed by service s_{ij}. The scheduling algorithm has to route the data along the paths in such a way to obtain the shortest workflow execution time and meet the cost constraint. The larger the budget that can be spent on the services

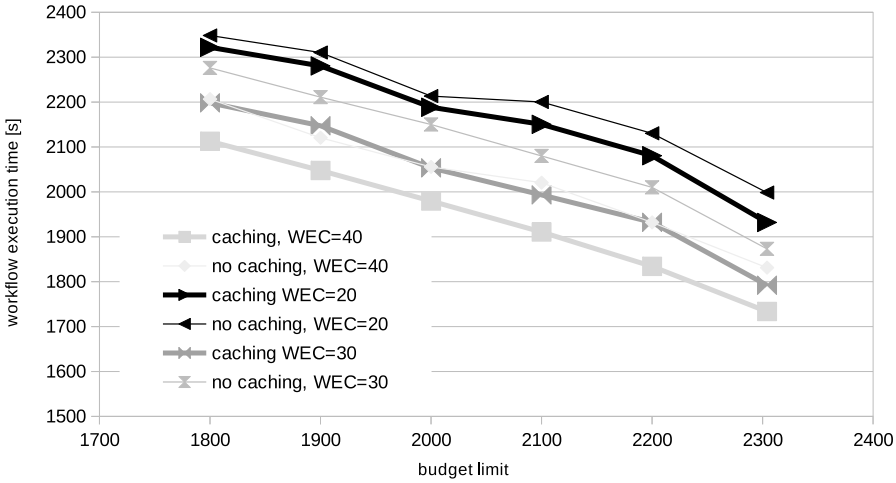

Fig. 13 Workflow execution time vs. storage costs

the faster services can be selected, i.e., more data packets can be routed to faster services. Following the results of the previous experiments, the streaming mode was set for efficient parallel computing and a small size of a data packet equal to 5 files to increase the streaming throughput and shorten the workflow execution time.

Firstly, before allocation of the initial budget among services and costs, Fig. 13 presents the workflow execution time for various combinations of three parameters:

- the BeesyCluster storage space (WEC),
- enabled or disabled caching of output images from one service when the next service is not yet ready for accepting new images,
- various cost bounds on the total cost of selected services.

Obviously, the following facts can be noted:

(1) Increasing the BeesyCluster storage space for the workflow execution engine results in shorter workflow execution times.
(2) Caching of output images from one service when the next service is not yet ready for accepting new images results in noticeably better workflow execution time than without caching.
(3) Increasing the cost bound on the services results in shorter workflow execution time as faster services can be selected.

Now, the goal is to determine whether various sets of costs for storage and services require different distribution of the budget among these components to obtain shorter workflow execution times.

Let us now assume that c_{WEC} is the cost of storage of a unit of data (represented by an input file to a service that defines a chunk of data to be processed) in the BeesyCluster workflow execution system. The following configurations can be considered.

$B = 2500$ *and* $c_{\mathrm{WEC}} = 10$ As an example, let us assume that the user has a budget equal to 2500 and $c_{\mathrm{WEC}} = 10$ (for storage of one file, i.e., the cost of storing 10 files

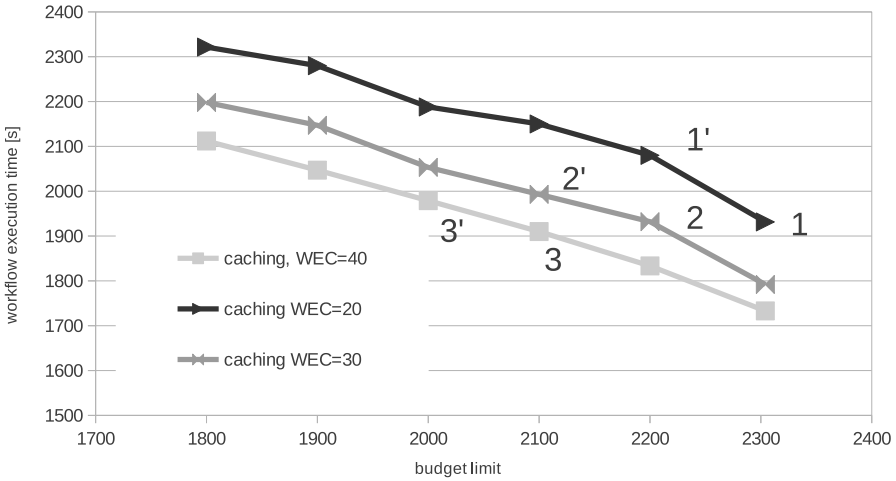

Fig. 14 Workflow execution time vs. storage costs, $c_{\mathrm{WEC}} = 10$

in the BeesyCluster management system is 100). This allows various distributions of the cost among B_{WEC} and costs of services. Some of these are shown in Fig. 14:

(1) $B_{\mathrm{WEC}} = 200$ for storage of 20 files in the system,
(2) $B_{\mathrm{WEC}} = 300$ for storage of 30 files in the system,
(3) $B_{\mathrm{WEC}} = 400$ for storage of 40 files in the system.

Figure 14 shows these cases from which one can determine that the lowest execution time is given by configuration 3.

$B = 2400$ *and* $c_{\mathrm{WEC}} = 10$ The second case we consider is with the initial budget of 2400 for which one can see equivalent distributions as $1'$, $2'$, and $3'$ out of which $3'$ gives the best result.

$B = 2700$ *and* $c_{\mathrm{WEC}} = 20$ The third case we consider is with the budget equal to 2700 and $c_{\mathrm{WEC}} = 20$. The latter means that storage of one file costs 20, and consequently the cost of storing 10 files in the BeesyCluster management system is 200. In this case, some of possible configurations are (Fig. 15):

(1) $B_{\mathrm{WEC}} = 400$ for storage of 20 files in the system,
(2) $B_{\mathrm{WEC}} = 600$ for storage of 30 files in the system,
(3) $B_{\mathrm{WEC}} = 800$ for storage of 40 files in the system.

This time the best configuration is 1 and apparently it does not pay off to invest into the storage of the execution engine if it is too expensive.

To conclude, if a workflow application is to be run many times, which is very often the case with scientific workflows, it should be preceded with analysis as presented above to make sure that the budget is properly distributed among intermediate storage space and the services depending on relative costs.

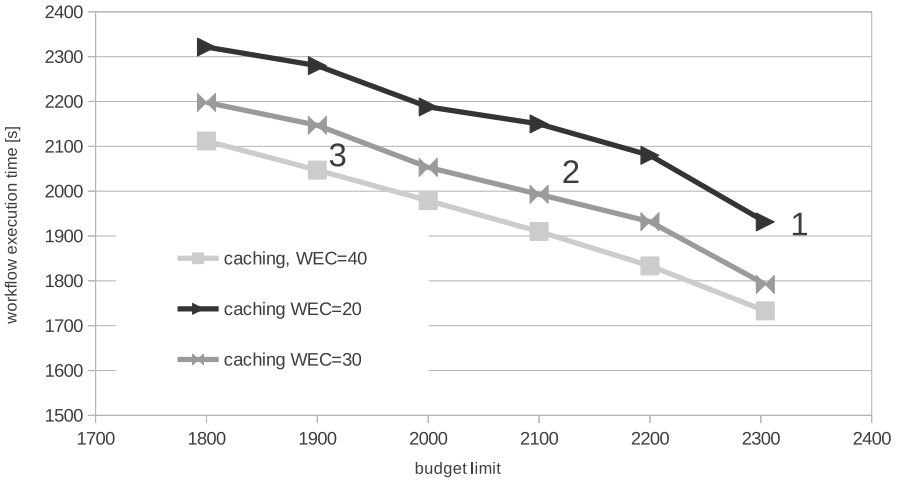

Fig. 15 Workflow execution time vs. storage costs, $c_{\text{WEC}} = 20$

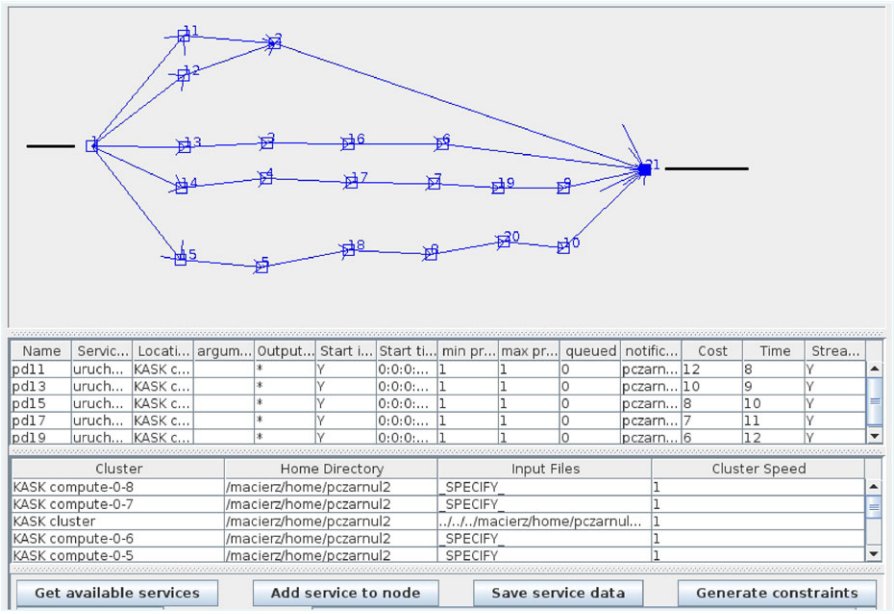

Fig. 16 Irregular testbed workflow

5.3.3 Irregular workflow application

Additionally, the implemented workflow execution engine with storage constraints and data caching was tested on a more irregular (in terms of the length of parallel paths and synchronization) acyclic workflow application shown in Fig. 16. Streaming was enabled, $ds_{ij}^{\max} = 10$ files, PARDS $= 1$, SDP $= 1$ file, various numbers of input

Table 2 Configurations and results for an irregular workflow application

Configuration	Workflow execution time (s)
16 input files, data caching, WEC $= \infty$	666
16 input files, no data caching, WEC $= \infty$	682
16 input files, data caching, WEC $= 2$	705
16 input files, no data caching, WEC $= 2$	725
80 input files, data caching, WEC $= \infty$	2667
80 input files, no data caching, WEC $= \infty$	2860
80 input files, data caching, WEC $= 5$	2701
80 input files, no data caching, WEC $= 5$	3004

files of 12–13 MB each and various setting were used as indicated in Table 2 along with corresponding execution times. Each task was assigned 5 services with cost/time ratios as follows: 12/8, 10/9, 8/10, 7/11, and 6/12. The results have confirmed that differences can be observed for various settings proving that the engine works as intended.

6 Summary and future work

The paper presented an extended model of scheduling workflow applications with optimization that incorporates not only the standard QoS parameters such as service cost and execution time but also storage constraints and storage costs. Storage sizes of data on resources from which services fetch and to which write input/output are considered. Compared to other works, the paper incorporated into the model and investigated also storage limitations for the workflow execution engine. Data caching for workflows with data streaming was proposed and implemented. Standard scheduling and workflow execution engines may not be able to run workflows with such storage constraints if, e.g., services are offered from commodity servers with small disk storage. The proposed solution provides an engine that copes with such environments perfectly.

Since storage space entails costs and impacts performance, the paper presented a method to distribute the budget for running a workflow not only among services, but also the storage for the execution engine.

The paper presents performance tests for various sizes of data packet when processing data in streams and for various storage capacities of the execution engine. The author has implemented a workflow execution engine that incorporates all the presented solutions and performed tests for a digital image processing workflow. The engine, deployed in the BeesyCluster middleware, is used for running large scale workflow applications as well as for teaching HPC and distributed systems at Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Poland.

It was shown in Sect. 5 that considering storage is important for the basic structures such as fork, parallel paths, and consequently join at the end of each workflow

application. Such constructs (fork, parallel paths, join) do appear in larger workflows modeled as acyclic directed graphs. The latter (as considered in this work) will naturally consist of parallel tasks being synchronized at various points of the workflow. This means that seeing differences between consideration of storage or not on the tested structures (and this has been shown) will also be visible for larger workflow applications that will inevitably contain such structures. Additionally, a workflow application with a more irregular structure was tested. For two different input data sets, it was confirmed that the proposed features, i.e., data caching and considering storage constraints do affect the workflow execution time.

For future work, Oceanstore or similar distributed storage systems can be considered as a means of potential handling of data when running thee workflow. Currently, this step is handled by BeesyCluster in case of the centralized execution engine or distributed agents in case of the distributed engine [12]. So, in this respect, such a system could act as a much larger capacity storage for data copied between locations on which services are installed. This can lead to potential higher limits on what the WEC parameter represents in the current model and will be used in future versions of the system.

References

1. Kepler user manual (2008) Version 1.0.0. https://code.kepler-project.org/code/ke-pler-docs/trunk/outreach/documentation/shipping/UserManual.pdf
2. Abrishami S, Naghibzadeh M, Epema DH (2012) Cost-driven scheduling of grid workflows using partial critical paths. IEEE Trans Parallel Distrib Syst 23:1400–1414. http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.303
3. Aggarwal R, Verma K, Miller J, Milnor W (2004) Constraint driven web service composition in meteors. In: Proceedings of IEEE international conference on services computing (SCC'04), pp 23–30
4. Aggarwal R, Verma K, Miller J, Milnor W (2004) Dynamic web service composition in meteors. Technical report, LSDIS Lab, Computer Science Dept, UGA
5. Bharathi S, Chervenak A (2009) Scheduling data-intensive workflows on storage constrained resources. In: Proceedings of the 4th workshop on workflows in support of large-scale science, WORKS'09. ACM, New York, pp 3:1–3:10. http://doi.acm.org/10.1145/1645164.1645167
6. Blythe J, Jain S, Deelman E, Gil Y, Vahi K, Mandal A, Kennedy K (2005) Task scheduling strategies for workflow-based applications in grids. In: CCGrid 2005. IEEE international symposium on cluster computing and the grid, vol 2, pp 759–767
7. Buyya R (ed) (1999) High performance cluster computing. Architectures and systems. Prentice Hall, New York
8. Canfora G, Penta MD, Esposito R, Villani ML (2005) An approach for qos-aware service composition based on genetic algorithms. In: Proceedings of the 2005 conference on genetic and evolutionary computation, GECCO'05. ACM, New York, pp 1069–1075. http://doi.acm.org/10.1145/1068009.1068189
9. Chin SH, Suh T, Yu HC (2010) Adaptive service scheduling for workflow applications in service-oriented grid. J Supercomput 52(3):253–283. doi:10.1007/s11227-009-0290-9
10. Czarnul P (2010) Modeling, run-time optimization and execution of distributed workflow applications in the jee-based beesycluster environment. J Supercomp, 1–26. doi:10.1007/s11227-010-0499-7
11. Czarnul P (2010) Modelling, optimization and execution of workflow applications with data distribution, service selection and budget constraints in BeesyCluster. In: Proceedings of 6th workshop on

large scale computations on grids and 1st workshop on scalable computing in distributed systems, international multiconference on computer science and information technology, pp 629–636. IEEE Catalog number CFP0964E
12. Czarnul P, Matuszek MR, Wójcik M, Zalewski K (2011) Beesybees: a mobile agent-based middleware for a reliable and secure execution of service-based workflow applications in beesycluster. Multiagent Grid Syst 7(6):219–241
13. Foster I, Kesselman C, Nick J, Tuecke S (2002) The physiology of the grid: an open grid services architecture for distributed systems integration. In: Open grid service infrastructure WG. Global grid forum. http://www.globus.org/research/papers/ogsa.pdf
14. Glatard T, Montagnat J, Lingrand D, Pennec X (2008) Flexible and efficient workflow deployment of data-intensive applications on grids with moteur. Int J High Perform Comput Appl 22:347–360. doi:10.1177/1094342008098067
15. Keahey K, Tsugawa M, Matsunaga A, Fortes J (2009) Sky computing. IEEE Internet Comput 13:43–51. http://doi.ieeecomputersociety.org/10.1109/MIC.2009.94
16. Kyriazis D, Tserpes K, Menychtas A, Litke A, Varvarigou T (2008) An innovative workflow mapping mechanism for grids in the frame of quality of service. Future Gener Comput Syst 24(6):498–511. doi:10.1016ij.future.2007.07.009. http://www.sciencedirect.com/science/article/B6V06-4P940GS-1/2/e94f93365addf9e83adaa967051d60e7
17. Pandey S, Karunamoorthy D, Buyya R (2011) Workflow engine for clouds. In: Cloud computing: principles and paradigms. Wiley, New York, pp 321–344. ISBN 978-0470887998
18. Project TG: Workflow language (xwfl2.0). http://gridbus.cs.mu.oz.au/workflow/2.0beta/docs/xwfl2.pdf
19. Ramakrishnan A, Singh G, Zhao H, Deelman E, Sakellariou R, Vahi K, Blackburn K, Meyers D, Samidi M (2007) Scheduling data-intensive workflows onto storage-constrained distributed resources. In: Proceedings of the seventh IEEE international symposium on cluster computing and the grid, CCGRID'07. IEEE Comput Soc, Washington, pp 401–409. doi:10.1109/CCGRID.2007.101
20. Sakellariou R, Zhao H, Tsiakkouri E, Dikaiakos MD (2007) Scheduling workflows with budget constraints. In: Gorlatch S, Danelutto M (eds) Integrated research in grid computing. CoreGrid series. Springer, Berlin
21. Singh G, Vahi K, Ramakrishnan A, Mehta G, Deelman E, Zhao H, Sakellariou R, Blackburn K, Brown D, Fairhurst S, Meyers D, Berriman GB, Good J, Katz DS (2007) Optimizing workflow data footprint. Sci Program 15:249–268. http://dl.acm.org/citation.cfm?id=1377549.1377553
22. Syslo MM, Deo N, Kowalik JS (1983) Discrete optimization algorithms. Prentice-Hall, New York
23. Wieczorek M, Hoheisel A, Prodan R (2009) Towards a general model of the multi-criteria workflow scheduling on the grid. Future Gener Comput Syst 25(3):237–256
24. Yuan Y, Li X, Sun C (2007) Cost-effective heuristics for workflow scheduling in grid computing economy. In: Proceedings of the sixth international conference on grid and cooperative computing, GCC'07. IEEE Comput Soc, Washington, pp 322–329
25. Yu J, Buyya R (2005) A taxonomy of workflow management systems for grid computing. J Grid Comput 3(3–4):171–200. doi:10.1007/s10723-005-9010-8
26. Yu J, Buyya R (2006) A budget constrained scheduling of workflow applications on utility grids using genetic algorithms. In: Workshop on workflows in support of large-scale science, proceedings of the 15th IEEE international symposium on high performance distributed computing (HPDC 2006). Paris, France
27. Yu J, Buyya R (2006) Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. Scientific programming journal. IOS Press, Amsterdam. ISSN: 1058-9244
28. Yu J, Buyya R, Ramamohanarao K (2008) Metaheuristics for scheduling in distributed computing environments. In: Workflow scheduling algorithms for grid computing. Metaheuristics for scheduling in distributed computing environments. Springer, Berlin. ISBN 978-3-540-69260-7. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.144.7107
29. Yu J, Buyya R, Tham CK (2005) Cost-based scheduling of workflow applications on utility grids. In: Proceedings of the 1st IEEE international conference on e-science and grid computing (e-science 2005). IEEE Comput Soc Press, Melbourne
30. Yuan D, Yang Y, Liu X, Chen J (2010) A cost-effective strategy for intermediate data storage in scientific cloud workflow systems. In: IPDPS. IEEE Press, New York, pp 1–12
31. Zeng L, Benatallah B, Dumas M, Kalagnanam J, Sheng Q (2003) Quality driven web services composition. In: Proceedings of WWW 2003, Budapest, Hungary