Using MinMax-Memory Claims to Improve In-Memory Workflow Computations in the Cloud

Shuibing He, Yang Wang, Xian-He Sun, *Fellow, IEEE*, and Chengzhong Xu, *Fellow, IEEE*

Abstract—In this paper, we consider to improve scientific workflows in cloud environments where data transfers between tasks are performed via provisioned in-memory caching as a service, instead of relying entirely on slower disk-based file systems. However, this improvement is not free since services in the cloud are usually charged in a "pay-as-you-go" model. As a consequence, the workflow tenants have to estimate the amount of memory that they would like to pay. Given the intrinsic complexity of the workflows, it would be very hard to make an accurate prediction, which would lead to either oversubscription or undersubscription, resulting in unproductive spending or performance degradation. To address this problem, we propose a concept of minmax memory claim (MMC) to achieve cost-effective workflow computations in in-memory cloud computing environments. The minmax-memory claim is defined as the minimum amount of memory required to finish the workflow without compromising its maximum concurrency. With the concept of MMC, the workflow tenants can achieve the best performance via in-memory computing while minimizing the cost. In this paper, we present the procedure of how to find the MMCs for those workflows with arbitrary graphs in general and develop optimal efficient algorithms for some well-structured workflows in particular. To further show the values of this concept, we also implement these algorithms and apply them, through a simulation study, to improve deadlock resolutions in workflow-based workloads when memory resources are constrained.

Index Terms—Minmax memory claim, in-memory caching, deadlock avoidance, memory constraints, workflow scheduling

1 Introduction

Due to the benefits of cloud computing with respect to its elasticity, small start-up and maintenance costs, and economics of scale, the interest in deploying scientific workflows in cloud platforms has been kept ever-growing over the past few years [1], [2], [3].

A scientific workflow generally consists of a set of data-dependent tasks, forming a *weighted directed acyclic graph* (DAG), also called *workflow graph*, to carry out a complex computational process. The nodes in the workflow graph represent tasks (e.g., executable or a script) that accomplish a certain amount of work in the workload, and edges denote the data channels used to transfer data volume from source node to target node. In a cloud-based workflow computation, the data channels are typically implemented via an external provisioned storage system (e.g., a file system), which could incur substantial disk I/O overhead that can dominate the execution times [4], [5].

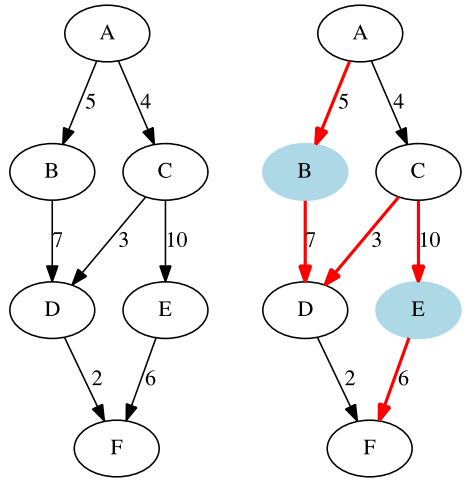
To address this issue, in-memory caching utility in the cloud provides an effective way, which aggregates massive memory resources across a dedicated cluster of servers to support all the data managements via a middleware software [6], [7]. With in-memory caching, the workflow computation could transfer data between tasks via fast, managed, in-memory caches, instead of relying entirely on slower disk-based file systems. Compared with disk read/write operations, this enhancement could come with potentially several orders of magnitude better end-to-end latency, and thus substantially improve the overall performance of the workloads.

Although the performance advantage of the in-memory cloud computing for scientific workflows is prominent, it does not come with no cost since services in the cloud are usually charged according to a "pay-as-you-go" fashion. As a consequence, the workflow tenants have to estimate the amount of caching memory that they would like to pay. Given the intrinsic complexity of the workflows, it would be very hard, if not impossible, for the workflow tenants to make accurate reservations on the resources to be used. Sub-optimal subscription would result in either unproductive spending (oversubscription) or performance degradation (undersubscription).

To address this problem, we propose a concept of *Min-Max-Memory Claim* (MMC) in this paper, which is defined as the **min**imum required memory resources for the workflow computation without compromising its **max**imum task concurrency (also the performance), The MMC is desirable for cost-effective computing in the cloud because the amount of

- *S. He is with the State Key Laboratory of Software Engineering, Computer School, Wuhan University, Luojiashan, Wuhan, Hubei 430072, China, and the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Xueyuan Blvd. 1068, Shenzhen 518055, China. E-mail: heshuibing@whu.edu.cn.*
- *Y. Wang is with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Xueyuan Blvd. 1068, Shenzhen 518055, China. E-mail: yang.wang1@siat.ac.cn.*
- *X.-H. Sun is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616. E-mail: sun@iit.edu.*
- *C. Xu is with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China, and the Department of Electrical and Computer Engineering, Wayne State University, Detroit, MI 48202. E-mail: cz.xu@siat.ac.cn.*

Manuscript received 13 Apr. 2016; revised 18 Aug. 2016; accepted 25 Sept. 2016. Date of publication 28 Sept. 2016; date of current version 15 Mar. 2017. Recommended for acceptance by Z. Chen.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2614294



(a) Workflow graph (b) Minmax claim (31 memory units).

Fig. 1. An example of workflow graph and its minmax memory claim. The Nodes represent tasks and the weighted edges represent the memory resources used to create the data channels in the cloud for communication between tasks. In (b), A and C are finished, and B and E are concurrently running.

memory provisioned over the MMC threshold cannot make any performance contributions. As such, it is very beneficial to those who intend to have maximized performance while minimizing the budget for the workflow computation in the cloud.

Fig. 1a is an example workflow graph where the weighted edges represent the resources used to create memory channels for communication between tasks. To maximize the concurrency of the workflow, we have to satisfy a minimum memory requirement. Fig. 1b shows when tasks B and E execute concurrently with their input and output channels occupied to maximum, the MMC reaches 31 memory units (A and C are finished, and their allocated memory has been collected as it is no longer used by the subsequent tasks). Again, this value is the minimum memory capacity to ensure the maximum DOC of this particular workflow graph.

As for the MMC of workflow graph in general form, in this paper we provably reduce its computation to finding a *maximum weighted clique* in a general graph that is derived from the original workflow graph by applying certain transformations. To tackle the intractability of this problem [8], in this paper, we focus squarely on certain well-structured workflows in reality [9], [10], [11], and exploit their graph structures to design efficient optimal algorithms.

The MMC is not a fixed value, rather, it is monotonically decreased as more tasks in the workflow are finished during the computation. Therefore, in practice when multiple workflow instances execute concurrently, it is not necessary to allocate memory resources to each workflow instance in alignment with its maximum MMC, which could further reduce the cost. This dynamic computation of the MMC is particularly valuable for multi-tenant environments when multiple workflow instances from different tenants run concurrently on the shared caching memory. However, in this situation, deadlock is a minimal pragmatic concern in avoiding the performance inference between concurrent instances, and how to resolve it is also a challenge. For example, given resource budget of 31 units in Fig. 1, deadlock could happen if two workflow instances are executed concurrently and the memory resources are allocated in a misguided way. For instance, if 9 units are allocated to the first instance and 22 units are left to the second, in this case no one is sufficient to make progress, and both instances fall into a deadlock state.

As an example to show the value of the MMC in cost reduction and performance maximum in multi-tenant environments, we leverage the concept to extend the banker's algorithm to deadlock avoidance in multiple concurrent workloads. With simulation studies, we show that the proposed algorithms not only dominate the compared algorithms with respect to deadlock resolution but also exhibit some advantages over them to potentially improve overall workload performance.

In summary, we make the following contributions.

1) We propose a concept of minmax memory claim for computational workflow to quantify its minimum required memory resources for maximum concurrency and performance.
2) We present the procedure how to find the claims for those workflows with arbitrary graphs in general and develop optimal efficient algorithms for three selected representative workflows in particular.
3) We exploit the proposed concept to optimize the banker's algorithm for deadlock avoidance in workflow computation.
4) We implement the algorithms and apply them, through a simulation study, to deadlock avoidance in workloads that consist of a number of concurrent instances in the cloud.

The rest of the paper is organized as follows. We overview some related work in the next section and describe the computational model of workflow computation in Section 3. After that, we present our algorithms to MMC in Section 4 and its application in Section 5. We evaluate the proposed algorithms through a simulation study in Section 6, and conclude the paper with some future work in Section 7.

2 Related Work

There are a plethora of studies on the bounded memory computing in the realm of high-performance computing (HPC) systems, each being from different angles and adopting different approaches [12], [13], [14], [15], [16]. However, almost all of them, based on our overview, are oriented to those memory intensive programs in which most of the time is spent waiting for memory operations to complete, instead of the maximum required memory size, the concern in this paper. The main reason to account for this phenomenon, in our opinion, is simply the evolution of memory technology that renders the relevance of memory capacity for HPC applications gradually decreased. However, this situation is experienced a great change when considering HPC applications in the cloud whose compute resources are provisioned on demand as per "pay-as-you-go" billing model. Memory is a precious resource in cloud platforms, especially in clouds whose in-memory caching or computing is usually provisioned as a service [17]. Given this fact, memory resources, together with other compute resources (e.g., processors, storage, and networks), are always optimized to improve their utilization in an economic way via so-called

cost-effective scheduling algorithms, when deploying workflow computation in the cloud [18], [19], [20]. Although these studies are resource-centric, and improve its utilization in diverse contexts, no one delve into the problem from our angle to improve the workflow-based computation in the cloud with the concept of MMC.

There exist some infrastructures that could be used to facilitate in-memory computing in the cloud. Tachyon [21] is a memory-based file system with supports of re-computation technique to facilitate reliable data sharing across jobs. Similarly, Amazon ElastiCache [17] as an on-demand service can add an in-memory caching layer to compute infrastructure for performance optimization. Both techniques can provide workflow computations with efficient in-memory caching services even though each has its own overhead in doing so.

Computation and memory substrate coupled architectures, such as Spark/RDD [22] and GPI-Space [23], are promising for in-memory computation. Spark/RDD [22] is an open-source cluster computing framework that leverages its multi-stage in-memory primitives to achieve performance up to an order of magnitude faster compared with its disk-primitive counterpart (say Hadoop). GPI-Space [23] is a more recent example that is capable of doing all parallel computation in memory via a virtual memory layer to omit the higher latencies and performance bottlenecks of traditional I/O. Although these studies are resource-centric, and improve its utilization in diverse contexts, no one delve into the problem from our angle to improve the workflow-based computations in the cloud with the concept of MMC.

A study bearing a similarity in spirit to ours is BLAZE [6], [7], which is a simple multi-tenant data cache scheme designed to guarantee a minimum cache memory share for each tenant to boost concurrency. The MMC scheme goes a further step that targets the minimum share for the workflow workloads, but unlike BLAZE, it does not consider the proportional allocation of cache memory shares among multiple tenants. Chiu et al. [24] investigate elastic cloud caches for accelerating service-oriented computations where the cache system can be scaled up during peak querying times, and back down to save costs in other cases. In contrast, the cache memory considered in our case is not elastic, instead, it is determined by the MMC and fixed during the computation. However, with the proposed deadlock resolution, it can be fully utilized to support multiple concurrent workflow instances for cost saving.

Leveraging dynamic resource requirements to improve the overall resource utilization can be dated back to the early research to improve the banker's algorithm with respect to its effectiveness in safety checking [25], [26]. Lang looks at this problem in multiprocessor systems where the control-flow of a workflow is modeled as a rooted-tree-like resource-request graph. In Lang's algorithm, the dynamic maximum resource claim (e.g., MMC in our case) is approximated by the localized maximum requirements of the induced subgraph to be executed (i.e., region) [26] as opposed to the total requirements of the whole control-flow graph (i.e., the workflow). Lang's algorithm then uses this dynamic requirement as the localized maximum claim for each scheduled workflow instance to improve the banker's algorithm for deadlock avoidance. Although Lang's algorithm can be applied to memory resources, and compared to the banker's, is effective in resource utilization, it suffers from the limitation with respect to the structure of the resource-request graph and cannot be used for general *DAG-based* workloads.

As for those tasks with general resource-request graphs, Wang and Lu [27] develop two dataflow-based algorithms, namely *DAR* and *DTO*, for avoiding the deadlock issue in concurrent task executions where the data channels are created in an external file system. In *DAR*, the *minmax resource claim* for each task's safety check is upper bounded by aggregating the storage resource requirements, defined by the remaining tasks to be executed, while in *DTO*, the dataflow knowledge is exploited to compute the *dynamic* maximum claim by topologically ordering the tasks during the safety check. *DAR* can maximize the concurrency but it is not quite effective in resource utilization due to its conservativeness in (over-)estimating the localized maximum claim. In contrast, *DTO* can minimize the storage resources, but it is incapable of identifying the minimum resources to guarantee the concurrency.

As a consequence, our results can be viewed as an improvement to the existing findings from a perspective of shared in-memory caching service where workflows with general DAGs can maximize their performance with minimum memory resources without caring about the deadlock issues.

3 Computational Model

3.1 Workflow Model

We model a workflow as a *workflow graph*, a weighted DAG, $G(V, E)$, where V represents a set of nodes and E a set of edges. A node in the DAG represents a *task* which in turn allocates memory, performs computation, and then deallocate memory in a sequential order without preemption. The weight of a node $v \in V$ is called *computation cost*, denoted by $m(v)$. An edge $e_{ij} \in E$ represents a data channel for data delivery from node v to node $u, u \in Out(v)$, here $Out(v)$ is v's out-edge neighbors; its weight $w(e_{ij})$ indicates the size of the data volume. The communication between nodes could be achieved by following the *producer-consumer model*, that is, if node v needs to communicate with node $u \in Out(v)$, it (producer) writes the data to the corresponding data channel, which will be read by node u (consumer). As such, each node v is associated with a *read set* and a *write set*, denoted respectively as $R_v = \{r_v^1, \ldots, r_v^k\}$ and $W_v = \{w_v^1, \ldots, w_v^l\}$. The *claim* on the required data channel resources by node v_i is, therefore, known a prior and can be computed as $b_v = \sum_{j=1}^{k} |r_v^j| + \sum_{j=1}^{l} |w_v^j|$ where $|r|$ and $|w|$ indicate the sizes of the set r and w, respectively.

The precedence constraints of a DAG dictate that the nodes are strict with respect to both their inputs and their outputs in the sense that a node cannot begin execution until all its inputs have arrived and no outputs are available until the node has finished and at that time all outputs are simultaneously accessible to its destination nodes. The node and edge weights as well as the shape of the workflow graph are determined by application and not changed during the computation. In this research, for performance optimization, we are particularly interested in leveraging the in-memory caching as a service to implement the data channels, instead of using the underlying file system as did

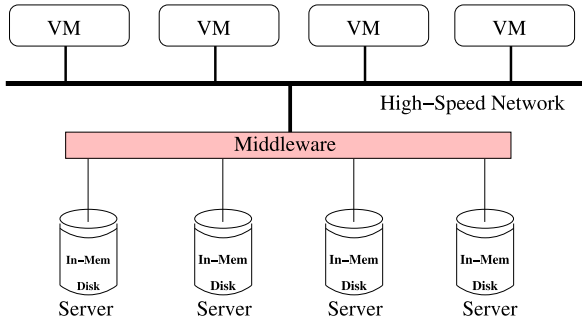


Fig. 2. Cluster architecture in the cloud for workflow computation. The middleware is used to construct a globally shared in-memory caching service by aggregating the collection of in-memory caching servers.

in traditional method. Thus, the effective use of the service in an economical way is always a concern.

Without loss of generality, a single source task and a single sink task are assumed in the workflow graph.[1] These two nodes can be viewed as the tasks in the graph that stage in the initial inputs and stage out the result outputs, respectively. Also, given the requirements that the memory channel resources are reclaimed immediately whenever they are no longer used by the subsequent nodes, the net amount of memory after the workload computation is zero. Hereafter, we use the terms node and task interchangeably.

3.2 Execution Model

Our execution model is built on top of a virtual cluster as shown in Fig. 2, which consists of a collection of virtual machines (aka. nodes) that have been configured to act like a traditional cluster. This typically involves installing and configuring job management software, such as a batch scheduler, and a shared storage system (e.g., network/distributed file system) [3]. In our particular context, the shared storage network of nodes (possibly disjoint from the network of compute nodes) is composed of in-memory caching storage nodes that are aggregated via the middleware to provide a globally shared in-memory caching service.

During the execution of a workflow instance, the life cycle of a task may experience several states. Initially, all the tasks in the workflow instance are in *blocked* state. A task becomes *free* if it has no parent tasks or all its parent tasks have been completed. Every free tasks can be scheduled but only those who have memory resources to accommodate their outputs enter the *ready* state for execution. Otherwise, they will be in *pending* state waiting for the availability of the memory resources. Of course, as soon as the required memory resources are available and also the MMC-based deadlock resolution is successfully triggered and completed, the tasks in *pending* state can be changed to *ready* state for execution again. The tasks in the *running* state are never stopped until they complete the computation. After a task has completed, it enters *done* state. A completed task will release the memory resources occupied by its inputs only which can be reclaimed for other tasks' executions, but keeping the memory resources as memory channels for the outputs used by the later tasks.

1. Any workflow can have such tasks used for data staging in and staging out, respectively.

Our model is deterministic, at least to the extent that the time, memory resources required by any task as well as the data dependencies among the tasks are pre-determined and remain unchanged during the computation as well.

4 MinMax-Memory Claim Algorithms

4.1 Basic Ideas

The basic idea of the proposed algorithms is first to augment the workflow graph with an *edge-node transformation* and compute the *Maximum Weighted Concurrent Set* (MWCS) of the *augmented workflow graph*, then provably show the MWCS is the MMC of the original workflow graph. The weight of the MMC could be further used as the maximum claims of the remaining tasks in the instance for deadlock avoidance. The MMC-based algorithm is less conservative in memory utilization than the existing ones [27], [28], but suffering from intractability as it is equivalent to finding a *Weighted Maximum Clique* (MWC) in a *derived graph* built from the augmented graph, and thus losing efficiency in the deadlock resolution.

To deal with the intractability and also show the values of the MMC concept, we select two commonly used workflow graphs in scientific computation, *lattice* and *fork&join*, and design efficient algorithms to compute their optimal MMCs. With these algorithms, we further improve the classic banker's algorithm for deadlock avoidance among concurrent workflow instances in the cloud. The results can be also easily extended to other workflows with regular shapes.

4.2 MinMax Memory Claim

4.2.1 Basic Definition

At any time instance t during the computation, we can classify the nodes in the graph into three groups:

(1) *Done(t)*: the nodes that have been completed prior to t.
(2) *Running(t)*: the nodes in $V - Done(t)$ that are running concurrently at t
(3) *Blocked(t)*: the remaining nodes at t, i.e., $V - (Done(t) \cup Running(t))$.

Accordingly, we can define the amount of memory resources that held in each set of nodes as follows:

$$\varphi_{d\rightarrow r}(t) = \sum_{v_i \in Done(t), v_j \in Running(t)} w(e_{ij}), \quad (1)$$

$\varphi_{d\rightarrow r}(t)$ is the total memory resources that are created by $Done(t)$ and being used by $Running(t)$ at t

$$\varphi_{d\rightarrow b}(t) = \sum_{v_i \in Done(t), v_j \in Blocked(t)} w(e_{ij}). \quad (2)$$

Similarly, $\varphi_{d\rightarrow b}(t)$ is the total memory resources that are created by $Done(t)$ and will be used by $Blocked(t)$ at t

$$\varphi_{r\rightarrow b}(t) = \sum_{v_i \in Running(t), v_j \in Blocked(t)} w(e_{ij}). \quad (3)$$

Finally, $\varphi_{r\rightarrow b}(t)$ is the total memory resources that are created by $Running(t)$ and will be used by $Blocked(t)$ at t. As a consequence, the *Minmax Resource Claim* at time t is determined by the maximum of $\varphi_{d\rightarrow r}(t) + \varphi_{d\rightarrow b}(t) + \varphi_{r\rightarrow b}(t)$,

which is also the minimum memory resources to ensure the maximum concurrency of the workflow. Thus, we define the MMC of the workflow graph $G(V, E)$ as

$$MMC(G) = \max_{t \in [0,l]} \{\varphi_{d \to r}(t) + \varphi_{d \to b}(t) + \varphi_{r \to b}(t)\}, \quad (4)$$

where l represents the makespan of the workflow instance. An example of these definitions is shown in Fig. 1b where at a particular time t, `Done(t)={A,C}`, `Running(t)={B,E}`, and `Block(t)={D,F}` Furthermore, $\varphi_{d \to r}(t) = e_{AB} + e_{CE} = 15$, $\varphi_{d \to b}(t) = e_{CD} = 3$, and $\varphi_{r \to b}(t) = e_{BD} + e_{EF} = 13$. Then `MMC(G) = 15 + 3 + 13 = 31`.

4.2.2 Our Solutions

Intuitively, we can identify the *Maximum Weighted Concurrent Set* of the (directed) graph and aggregate its weights as the MMC. However, this method is not always correct as some memory channel resources which will be used in the subsequent tasks could not be counted, e.g., in Fig. 1b, the memory channel between nodes `C` and `D` (i.e., e_{CD}) could be missed and select the concurrent nodes `B` and `C` as the MMC.

Alternatively, we can leverage the weighted nodes to compute the maximum time-frame, from the earliest start to the latest end, for each memory channel that needs to be maintained for the subsequent computation, and then sweep along the time axis to accumulate the memory channels as the MMC. Unfortunately, this strategy is still incorrect as the workflow graph is stretchable along the time axis during the computation, which leads to the inaccuracy of the sweep approach.

Therefore, to address this problem, we have to find other solutions. The basic idea of our approach is similar to the one that finds the MWCS in the workflow graph, but with an extension to count the weights of those edges that span across the frontier of the concurrent task set (e.g., $\varphi_{d \to b}$) as well. To this end, we first augment the workflow graph via an edge-node transformation, and then find its MWCS, which is equivalent to finding the *Maximum Weighted Clique* in a *derived graph* created from the augmented graph.

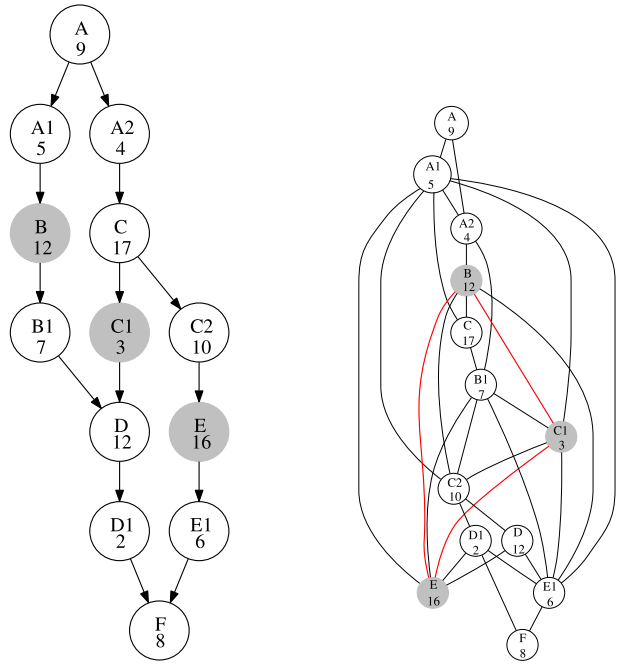
Given a workflow graph $G(V, E)$, we define an *augmented workflow graph* $G'(V', E')$ with the following *edge-node transformations:*

1) for each node v in $G(V, E)$, define $w(v) = b_v$;
2) for each edge e in $G(V, E)$, add a dummy node v_e on the edge with a weight defined as $w(v_e) = w(e)$, and in the meanwhile, clear the edge weight (i.e., the augmented workflow graph is only node weighted).

This transformation can be completed within $O(|V| + |E|)$ time to create $G'(V', E')$ where $V' = V \cup \{v_e | e \in E\}$ and $E' = \{e' | e' = (u, v_e) + (v_e, v)\ where\ e = (u, v) \in E\}$. With this transformation we also clearly have $|V'| = |V| + |E|$ and $|E'| = 2|E|$.

An example of augmented workflow graph is shown in Fig. 3a where the workflow graph in Fig. 1a is augmented. The essence of this transformation is to convert finding *MWCS* with respect to nodes to finding it with respect to edges by treating each edge as a weighted dummy node in the augmented graph.

With the augmented workflow graph, we have the following theorem,



(a) Augmented DAG graph $G'(V', E')$ (b) Derived graph $G''(V'', E'')$

Fig. 3. An example of workflow augmented DAG of Fig. 1a and its derived graph for computing the maximum weight clique. The nodes with subscripts are dummy nodes, such as A1. MWCS = {B,C1,E} and MWC = {B,C1,E}.

Theorem 4.1. *The weight of the MWCS of the augmented workflow graph is equal to the MMC of the original workflow graph, i.e.,* $\mathcal{W}(MWCS(G')) = MMC(G)$.

Proof. The proof is straightforward as for the weighted concurrent set $WCS(t)$ at time instance t in $G'(V', E')$, its occupied memory resources must belong to some active tasks which are running concurrently (i.e., $\varphi_{d \to r}(t) + \varphi_{r \to b}(t)$) or certain memory channels that have been created for later uses (i.e., $\varphi_{d \to b}(t)$). As such, we have

$$\mathcal{W}(WCS(t)) \leq \varphi_{d \to r}(t) + \varphi_{d \to b}(t) + \varphi_{r \to b}(t).$$

By applying the similar logic, the reverse argument is also correct, i.e., the size of occupied memory resources by concurrent running tasks and memory channels at t is not greater than the weight of $WCS(t)$.

Overall, there is one-to-one mapping between the occupied memory resources of the original workflow graph $G(V, E)$ and the weight of concurrent set in the augmented graph, $G'(V', E')$ i.e.,

$$\mathcal{W}(WCS(t)) = \varphi_{d \to r}(t) + \varphi_{d \to b}(t) + \varphi_{r \to b}(t), \quad (5)$$

whereby we can conclude the theorem by maximizing both sides of Eq. (5) in terms of t over the workflow makespan. □

The main ingredient of our algorithm is to compute *MWCS* on G', which is equivalent to finding MWC in a *derived graph* $G''(V'', E'')$ obtained by following transformations on the augmented graph $G'(V', E')$:

1) given $G'(V', E')$, we first compute its all-pairs of paths via *graph traversal* within $O(|V'| + |E'|)$ time and $\Theta(|V'| + |E'|)$ space where the length of the path

TABLE 1
Three Typical Workflow Applications: *Gromacs*, *Proteome Analyst* and *Bronze Standard Medical Imaging*

Application	Function	Shape
Gromacs [9]	A molecular dynamics simulation package to simulate the Newtonian equations of motion for systems with hundreds to millions of particles	pipeline
Proteome Analyst (PA) [10]	A bioinformatics tool to predict protein properties such as the general function and the subcellular localization of proteins using machine learning techniques	fork&join
Bronze Standard Medical Imaging (BSMI) [11]	A data intensive medical image processing application developed to overcome the difficulties of evaluating the accuracy and robustness of image processing algorithms when the reference image is not available	lattice

is defined as the total number of hops along the path. However, if there is no directed path between the pair, the length is $+\infty$.

2) based on the results of Step 1, we derive an undirected graph $G''(V'', E'')$ from $G'(V', E')$ by adding an undirected edge between any pair of nodes with distance $+\infty$ in $G'(V', E')$, and then remove all directed edges in $G'(V', E')$, which could be finished in $O(|V'|^2)$,
3) compute the MWC of $G''(V'', E'')$ as the *MWCS* of $G'(V', E')$.

The first step is to identify those pairs of nodes that can be executed concurrently. This step is only executed once for each graph with in. Note that the concurrency relationship is not transitive, so we have to search the MWC in the derived graph to ensure that all the selected clique nodes can be executed concurrently in the augmented workflow graph. Fig. 3b shows an identified MWC in the corresponding derived graph of Fig. 3a whose weight is 12 + 3 + 16 = 31.

Finding MWC in a general graph is a well studied problem, and it can be optimally solved by some existing algorithms [29], [30]. However, this problem is a typical NP-hard problem [8], and its optimal algorithms are not efficient. Therefore, either heuristics or optimal algorithms for some special graphs are always used in reality. In this study, we adopt the later strategy with focus on three special commonly-modeled workflows in scientific computations.

4.2.3 Optimal Algorithms for Selected Workflows

We study three representative workflows in molecular dynamics [9], bioinformatics [10], and medical image processing [11], each workflow could be modeled by *pipeline*, *fork&join*, and *lattice*, respectively as described in Table 1. The key point of this study is to transform the workflow graphs and exploit the features of the transformed graphs to compute their MWCSs in an efficient way. This computation could form the basis for computing other workflows that can be decomposed into the studied shapes. The SciEvol workflow for molecular evolution reconstruction can be viewed as a pipeline workflow followed by a fork&join workflow to infer evolutionary relationships on genomic data [31].

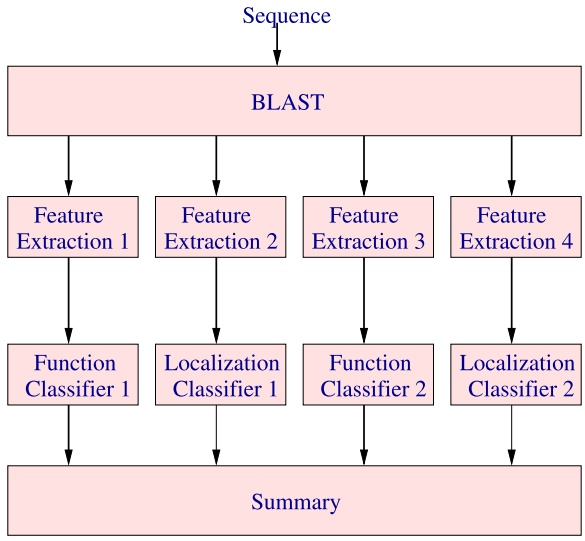


Fig. 4. An example of a typical PA workflow chart.

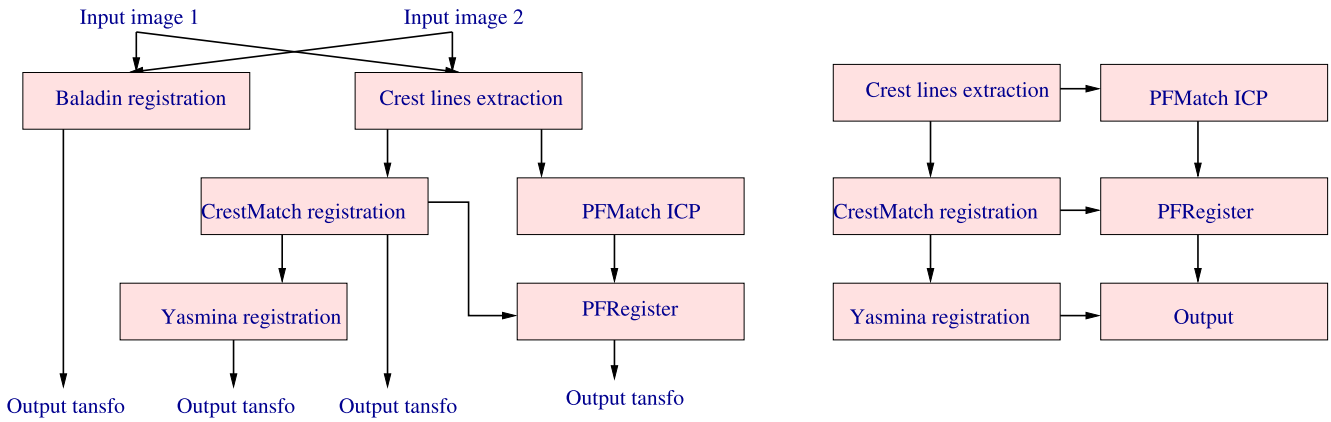
The selected *Proteome Analyst* (PA) [10] can be modeled as a fork&join workflow as shown in Fig. 4, The workflow first accepts a *proteome*[2] in the form of a text string, and then uses *BLAST* [32] to find the *homologs* among known proteins for each given protein. During this process PA also gains information about InterPro[3] families, which can also provide information about homology. PA uses this information to predict the classes of proteins. More specifically, the feature extraction programs (i.e., *Feature Extractions* in Fig. 4) take the homologs as the input and use different algorithms to extract some keywords or annotations as features. The extracted features are classified by different trained *classifiers* to determine the function and the localization for each query sequence within the cell. Finally, the program *Summary* gathers, summarizes and presents the outputs from various classifiers.

As described, the PA workflow is characterized by the number of stages and fan-out factors, exhibiting near-constant degree of concurrency (DOC) and can be taken as a representative of a large class of problems with a pipeline of parallel phases, including MapReduce workflows and other applications [9], [10]. Computing the MWCS for the fork&join graph is relatively easy since its augmented graph can be transformed into a rooted tree by u, denoted as $\mathcal{W}(u)$, and then solved by recursion 6 as follows in linear time and space complexity to aggregate the maximal weights of concurrent nodes in each sub-tree:

$$\mathcal{W}(u) = \max\left\{w(u), \sum_{v \in \mathcal{N}(u)} \mathcal{W}(v)\right\}, \tag{6}$$

2. A blend of proteins and genome that is often used to describe the entire complement of proteins expressed by a genome, cell, tissue or organism.

3. InterPro is an integrated documentation resource for protein families, domains and functional sites.



(a) Original workflow graph (b) Abstracted workflow graph

Fig. 5. An example of a bronze standard medical imaging workflow chart (a). We view it as a lattice-like workflow shape (b) in our discussion.

where node u is the root of the (sub-)tree with weight of $w(u)$ and $\mathcal{N}(u)$ is the child set of node u.

The medical image processing workflow (BSMI) [11] is a data-intensive medical image processing application developed to overcome the difficulties of evaluating the accuracy and robustness of image processing algorithms when the *ground truth* (i.e., reference image) is not available. The workflow is assembled from a set of basic tools (i.e., jobs, see Fig. 5a), each having its own function to process the data, extract quantitative information and analyze results. The workflow can be simplified to be a lattice-like workflow shown in Fig. 5b.

To fully study the MMC on this structure, we extend it to a general lattice structure, characterized by its width and height, exhibits variable concurrency, where the concurrency increases initially to a maximum degree and then decreases progressively. A variety of dynamic programming algorithms and numerical computation workflows have a lattice structure [11], [33]. An example of (4×5) a general lattice workflow is shown in Fig. 6.

To compute MWCS in an augmented lattice $(n \times m)$, we model the problem by defining a weight matrix M as follows:

$$M = \begin{pmatrix} b_{00} & b_{01} & \cdots & \cdots & b_{0m} \\ b_{10} & b_{11} & \cdots & \cdots & b_{1m} \\ \cdots & \cdots & b_{ij} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ b_{(n-1)0} & b_{(n-1)1} & \cdots & \cdots & b_{(n-1)m} \end{pmatrix},$$

in which according to the model in Section 3, $b_{ij} = \sum_{s=1}^{k} |r_s| + \sum_{s=1}^{l} |w_s|, 0 \le i < n, 0 < j \le m$, which is the memory weight of node n_{ij}. Note that if b_{ij} is a dummy node created by connecting edge nodes, its weight is assumed to be $-\infty$ so that it will never be selected. For example, if nodes `7,9,3,13` are edge nodes, the weight of `node 8` is $-\infty$.

Based on the M, the MWCS of lattice $(n \times m)$ is equivalent to finding a set of $b_{ij}, 0 \le i < n, 0 < j \le m$ that satisfies

$$\max \sum_{0 \le i < n, 0 < j \le m} b_{ij}, \tag{7}$$

subject to the restrictions:

1) no more than one element of b can be selected for each raw and column;
2) if some b_{ij} is selected, the next element must be only selected from the matrix $M[0 \ldots i+1, j-1 \ldots m]$;
3) if b_{ij} is not selected, the next element must be only selected from the matrix $M[0 \ldots i+1, j \ldots m]$.

Clearly, these restrictions are resulted from the data dependencies in the lattice, and based on which, we have the following dynamic programming algorithm for the problem where $L(i,j)$ compute the optimal value of MWCS for $M[i \ldots n-1, 0 \ldots j]$,

$$\begin{cases} L(i,j) = \max \begin{cases} L(i,j-1) \\ b_{ij} + L(i+1,j-1) \\ L(i+1,j) \end{cases} \\ L(i,0) = \max_{i \le k < n} b_{k0} \\ L(n-1,j) = \max_{0 \le k < j} b_{n-1k} \quad 0 \le i < n, 0 < j < m. \end{cases} \tag{8}$$

The recurrence is easy to follow. If b_{ij} is selected, we have $L(i,j) = b_{ij} + L(i+1,j-1)$. Otherwise, if b_{ij} is selected, we have $L(i,j) = \max\{L(i,j-1), L(i+1,j)\}$. The optimal value of $L(i,j)$ is thus defined by whichever is large. By following this argument, we can find the optimal weight of MWCS for the lattice workflow at $L(0,m)$. Since there are only mn possible sub-problems, the time and space complexity of this algorithm is $O(mn)$ for augmented lattice $(n \times m)$. However, when considering it in terms of the

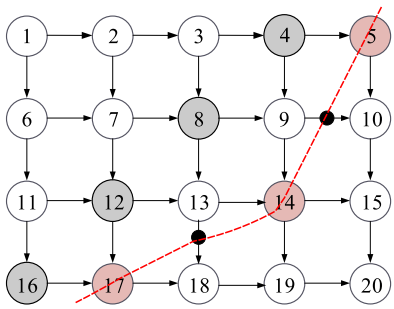


Fig. 6. An example of lattice (4×5) to show how its MWCS is computed. Nodes 17, 14, and 5, together with edges $e_{9,10}$ and $e_{13,18}$, consist of the MWCS.

original workflow graph, the time and space complexity is $O(|V| + 3|E|)$.

Fig. 6 shows a running example of how the MMC of a lattice (4×5) workflow is computed where tasks 17, 14, and 5, together with memory channels $e_{9,10}$ and $e_{13,18}$, consist of the MWCS of the augmented lattice.

Finally, the selected Gromacs workflow [9] can represent a class of pipeline workflows in reality whose structure can be viewed as a special case of fork&join (i.e., fan-out factor is one) or lattice (i.e., either width or height is one). The pipeline workflow is very common in scientific computations [34], [35], [36], its MMC is straightforward to compute.

5 Deadlock Avoidance: An Application of MMC

In this section, we leverage the results in the last section to extend the banker's algorithm for deadlock avoidance. Particularly, we make the only, but important improvement to the banker's algorithm by exploiting the MMC of the workflow dynamically to compute the localized maximum claim associated with each task, instead of, like in the banker's algorithm, statically computing such values in advance. With this improvement, compared to the algorithms in [27], [28], the proposed algorithm can dramatically improve the memory utilization.

Algorithm 1. MMC-Based Deadlock Avoidance Algorithm

```
1: procedure MCB(I_i, v_j^i)
2:                                   ▷ W_j^i and R_j^i are local variables.
3:   W_j^i ← getWriteSet(v_j^i)
4:   if (|W_j^i| > r(t)) then
5:         ▷ wait until there is enough free memory resources
6:     return false
7:   end if
8:                ▷ pretend to modify the system by assuming
9:                                   ▷ that v_j^i has completed.
10:  R_j^i ← getReadSet(v_j^i)
11:  r(t) ← r(t) − (|W_j^i| − |R_j^i|)
12:  alloc(i,t) ← alloc(i,t) + (|W_j^i| − |R_j^i|)
13:  updateNeed(need(i,t), I_i, v_j^i)
14:  if (SafetyCheck(I_i)) then                          ▷ O(|I|^2)
15:    r(t) ← r(t) − |R_j^i|
16:    alloc(i,t) ← alloc(i,t) + |R_j^i|
17:    return true                                       ▷ req safe
18:  else
19:    r(t) ← r(t) + (|W_j^i| − |R_j^i|)
20:    alloc(i,t) ← alloc(i,t) − (|W_j^i| − |R_j^i|)
21:    restoreNeed(need(i,t), I_i, v_j^i)
22:    return false                                      ▷ req unsafe
23:  end if
24: end procedure
```

5.1 MMC-Based Deadlock Avoidance Algorithm

Algorithm 1 shows our MMC-based deadlock avoidance (MCB) algorithm which is invoked when task v_j^i in workflow instance I_i is to be scheduled. In the algorithm, $r(t)$ is a global variable representing the available memory resources at t, $alloc(i,t)$ records the amount of memory resources that have been allocated to I_i, and $need(i,t)$ profiles the maximum claim of the memory resources for the remaining tasks in I_i after t. $alloc(i,0)$ is initialized to 0, and $need(i,0)$ is initialized to $max\_claim(i,0)$ for I_i, which is defined as $MMC_{t=0}(I_i)$. Both $need(i,t)$ and $alloc(i,t)$ are global variables in our algorithm.

MCB first checks if the current available memory resources are sufficient to satisfy the request of the task (obtained via *getWriteSet()*). If not, the task has to wait until sufficient memory resources are available. Otherwise, the task is assumed to be completed, and the corresponding data structures associated with the instance I_i (i.e., $r(t)$, $alloc(i,t)$ and $need(i,t)$) are updated accordingly. Subsequently, the safety of granting the request of task v_j^i is checked using the subroutine *SafetyCheck()* as in classic banker's algorithm.

Algorithm 2. The updateNeed Function

```
1: procedure UPDATENEED(need(i,t), I_i, v_j^i)
2:   graph ← getGraph(I_i)
3:   clearGraph(graph, v_j^i)                          ▷ O(|V| + |E|)
4:   augment ← getAugmentGraph(graph)
5:                                                     ▷ O(|V| + 3|E|)
6:   if (graph = fork&join||lattice) then
7:     Π ← getMWCS(graph)
8:     if (v_j^i ∈ Π) then
9:                          ▷ max_claim needs to be recomputed
10:      need(i,t) ← computeMMC(augment)
11:                                                    ▷ O(|V| + 3|E|)
12:    end if
13:  else                                  ▷ process general DAGs
14:    dr_graph ← getDerivedGraph(augment)
15:    Π ← getMWCS(dr_graph)
16:    if (v ∈ Π) then
17:                                                    ▷ by alg. [30], [37]
18:      need(i,t) ← computeMWC(dr_graph)
19:    end if
20:  end if
21: end procedure
```

The $updateNeed(need(i,t), I_i, v_j^i)$ is a key function in the MCB algorithm, which is used to update the need matrix of I_i by assuming v_j^i is finished. The pseudo code of this function is shown in Algorithm 2 where the time complexity of each major step is also listed. In the algorithm the workflow graph is first obtained and then cleared by $clearGraph()$ with two major operations. First, the weight of node v_j^i is set to zero (in $O(1)$ time) to ensure it to be never considered in the MMC computation. Second, when an incident edge (i.e., memory channel) to v_j^i is no longer used by its downstream tasks, set its weight to zero (in $O(|V| + |E|)$ time). After this the cleared graph is augmented (in $O(|V| + 3|E|)$ time), depending on the shape of the input workflow graph, the function uses different strategies to update the *need matrix* (in $O(|V| + 3|E|)$ time), either computing the optimal MMC as the updated *need matrix* for the selected fork&join or lattice graphs or obtaining the weight of the MMC, in upper bound or exact value, via some existing algorithms such as [30], [37]. Note that in both cases, the update is performed only if node v_j^i is in the existing MWCS of the augmented graph since only in this situation, the MWCS's weight could be changed.

The restore function performs reverse operations either on the augmented graph in a similar logic to recover the

need matrix. Note that to enable the algorithm to work in reality, we also need to empower the workflow scheduler with the functions similar to those in $clearGraph()$ to ensure that a finished task, together with its input memory channel if no longer used, is never involved in the subsequent computation. In this way, the maximum claim is localized and monotonically non-increasing (possibly decreasing) as the execution of the instance proceeds (i.e., dynamic computation, where $max\_claim(i,t)$ changes for different t).

As with the safety checking in the banker's algorithm, the *safety checking* algorithm in MCB iterates over all the concurrent instances in the workload and pools their allocated memory until either the need matrix of the current instance is satisfied (i.e., a safe state) or all instances are checked but it is impossible to complete the current instance (i.e., an unsafe state). This can be completed within $O(|I|^2)$ time where $|I|$ is the number of concurrent instances.

5.2 Algorithm Comparisons

We compare MCB with DAR in term of *domination relation*. DAR is selected as it is a similar deadlock avoidance algorithm designed for workflow-based workloads, which aggregates the caching memory resources of the tasks to be executed in the workflow instance as the maximum claim of the remaining graph for safety checking at run-time [27].

Definition (Domination Relation). *Given a system with a number of processes and a set of resources, for two different deadlock avoidance algorithms A and B, we define that A* dominates *B iff any safety decision made by B can be also made by A. We denote this relation by* $A \rightarrow B$.

Corollary 5.1. *MCB dominates DAR [27] with respect to the deadlock avoidance for workflow-based workloads when the memory resources are constrained, i.e.,* $MCB \rightarrow DAR$.

The corollary is straightforward as the maximum claim in *DAR* is computed by aggregating all the resource requests of the remaining tasks, which is not less than the MMC of the remaining graph.

6 Performance Studies

By following the tradition in the workflow studies, we adopt simulation-based methodology to investigate the performances of the proposed algorithms in terms of makespan and memory resource utilization on selected workflow workloads, each being composed of multiple instances. The simulated scheduler is built by using the discrete event simulation package SMURPH [38] according to the model presented in Section 3. It accepts the workflow instances from user submission. Each workflow instance, together with user's estimation of the memory resources for input and output data of each task, is exploited by a *deadlock resolver*, a core component of the scheduler, to compute the MMCs as its runtime maximum claims for deadlock avoidance in the computations.

6.1 Experimental Setup

In this section, we focus on the algorithm evaluations on the two selected workflow structures, fork&join and lattice, and leave other experimental work as on demand results, due to space limitation.

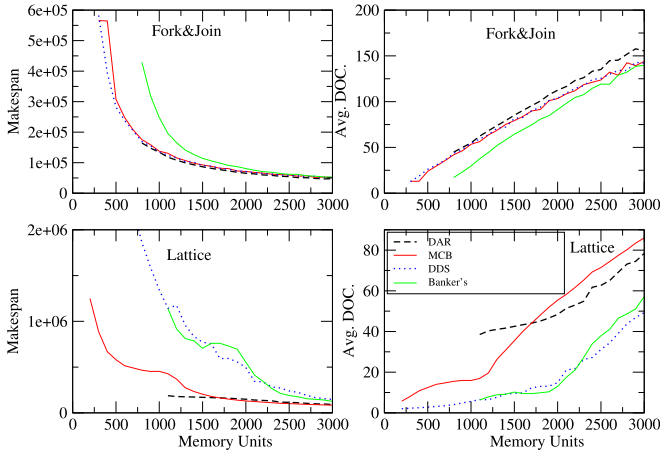


Fig. 7. Makespan comparisons between MCB and reference algorithms for fork&join and lattice workloads when the memory channel sizes are varied from 1 to 10 memory units.

Since we intend to study the performance of the proposed algorithms in general cases, except for the workflow graphs, we make no further assumptions on any *a prior* knowledge of the tasks such as their computation times, memory channel sizes, etc. As such, for all investigated workflow structures (i.e., fork&join (3×32) and lattice (8×12), we consider the task computation times to be uniformly distributed in $[500, 1{,}000]$ time units, while the memory channel sizes are uniformly distributed on $[1, 10]$ memory units. Additionally, we also assume that an unbounded number of compute nodes are available so that the maximum DOC would never be constrained by the compute nodes. This assumption is reasonable in our research since the cloud environments could virtually provide workflow computation with infinite resources, and on the other hand, we concern squarely with the minimum memory claim of workflow computation with respect to its maximum concurrency, which requires that the number of computational node should not be a restrictive factor. Finally, in each experiment, a total of 100 workflow instances are in each workload.

6.2 Simulation Results

In these experiments, we make a comprehensive performance evaluation of the proposed algorithms by comparing them with three reference algorithms, which bear the same property to maximize the degree of concurrency (DOC) with the available memory resources. The first is the banker's algorithm which is used as the baseline. The second is DAR, an extended banker's algorithm with an improvement to compute the maximum claim for each instance at runtime based on the aggregation of the memory requests of its remaining tasks [27]. The last one is a deadlock detection algorithm, called *DDS* [39], which is based on the detection and recovery principle to resolve the deadlock. Selecting DDS is only for performance reference as it is an alternative way to the deadlock.

Fig. 7 shows that for the fork&join, except for the banker's algorithm, the performance of all others is highly competitive with each other in terms of the makespan and DOC, which are all much better than the baseline. On the other hand, compared with *DAR*, *MCB* can work using much less memory

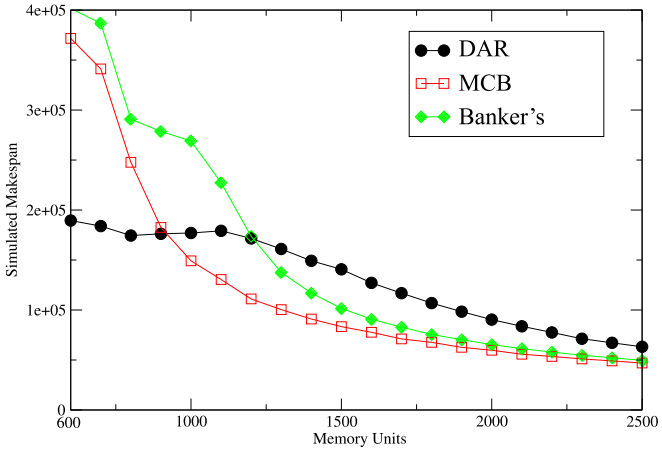


Fig. 8. Makespan comparison between MCB and reference algorithms for fork&join workloads with multi-readers.

resources, which even has the same performance behavior with *DDS*.

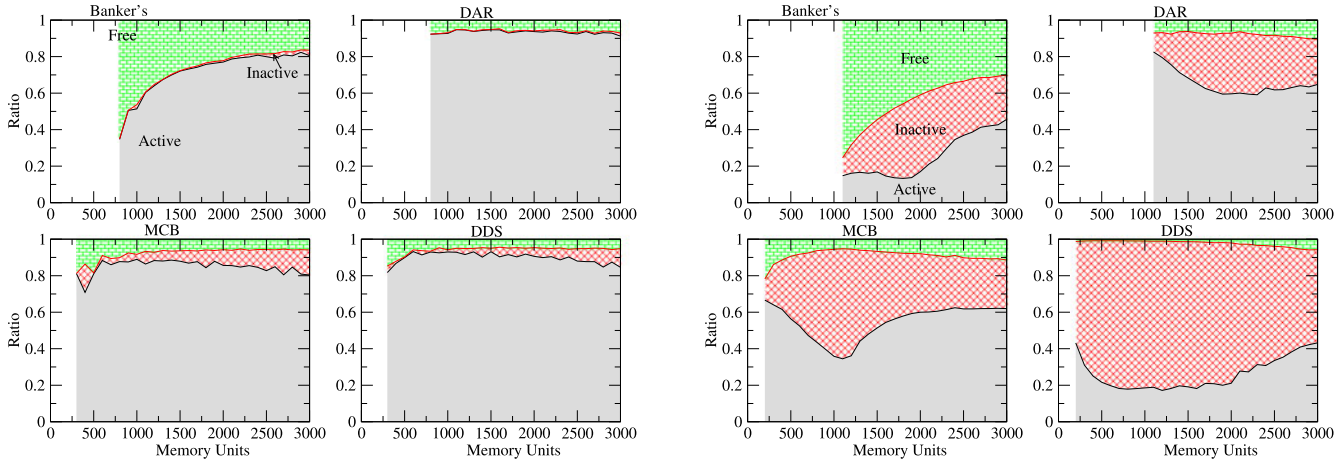
Unlike the fork&join, the lattice workload exhibits relatively low average DOC (Fig. 7). And also, the relative performances of these algorithms are changed significantly. First, the performance of *DDS* is degraded dramatically, especially when the memory is highly limited. This is because the number of instances admitted by DDS (no safety check), compared with other algorithms, is relatively high, increasing high resource contentions. Second, *DAR* is not always the performance leader, it is outperformed by *MCB* when the memory is moderate due to its conservativeness in the safety checking (say, 2,000 in our case, it is much clear if we zoom in the figure). *MCB* outperforms *DAR* by maximizing the average DOCs (Fig. 7). Third, again, like in fork&join, *MCB*, compared with other algorithms, can tolerate limited memory for the computation at cost of degraded performance since its maximum claim for each instance is much less than that of *DAR*.

When considering the impact of multi-readers, we allow the source task in fork&join to generate a single data file that is simultaneously read by multiple tasks via the memory caching service. By comparing with Fig. 7, we can observe the performance changes from Fig. 8 due to the impact of multi-readers. These changes are quite similar to what we have observed for the lattice workload. We then can follow the same arguments to account for these changes.

To validate our observations and further study these algorithms, we compare how each algorithm utilize the memory resources in term of *Memory Utilization*. To this end, at any time point during the computation, we classify the memory in the cloud into three classes. The first class includes the *free memory* that is not being used by any instance/task. The second class includes the memory that is being used by some active instances/tasks. We thus name it as *active memory*. Clearly, maximizing the active memory will lead to better memory utilization. The last class includes the remaining memory that is held by some inactive instances. Thus we call it *inactive memory*. Here, an active or inactive instance refers to whether the instance is running or not. Inactive instances cannot proceed due to insufficient memory, but their held memory (i.e., inactive memory) cannot be reclaimed for other instances. Therefore, inactive memory has adverse effects on the computations, which is different from the free memory that can be used whenever needed. Minimizing inactive memory can potentially improve the memory utilization.

For comparison purpose, we normalize the uses of three memory classes for both workloads by giving the ratio of each class and show their numerical results in Figs. 9a and 9b.

From Fig. 9a, for fork&join, we can observe that *DAR* has the smallest amount of free and inactive memory among the four algorithms whereas the banker's algorithm is in the worst case. This observation demonstrates that conservativeness in safety checking is not always detrimental to the overall performance of the workload since appropriate conservativeness could reserve more memory resources for those admitted instances to maximize their DOCs. This is also evidenced by the observation that the overall performance of *DAR* is better than that of *MCB*, even though it dominates *DAR* with respect to the deadlock resolution.



(a) Fork&join (3×32) (b) Lattice (8×12).

Fig. 9. The memory caching resources for fork&join workloads are broken down into three classes to specify how they are used during the computation: Free memory refers to the memory that is not being used by any instance/task; active memory is the memory that is being used by some active instances/tasks; in contrast, inactive memory means the memory is held by those inactive instances/tasks, which consists of the remaining memory.

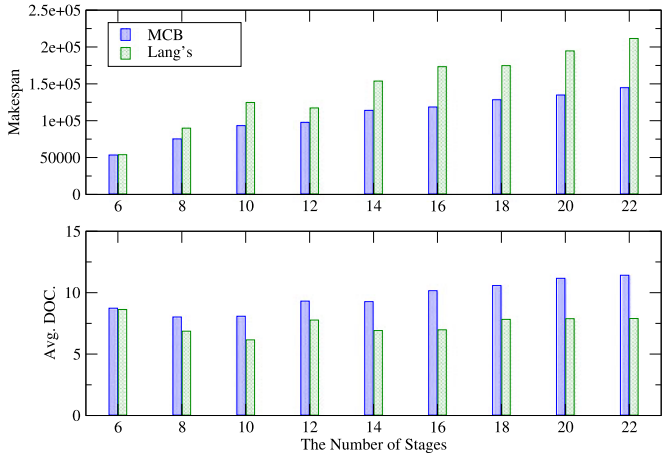


Fig. 10. Performance comparisons between different algorithms as the number of pipeline stages is varied from 6 to 22 (memory capacity is 200 units).

Additionally, we also found that the active memory ratio of *DDS* is slightly better than that of *MCB*. We attribute this phenomenon to the fact that *DDS* admits instances to execute without safety checking, which may lead to a higher active memory ratio. However, this higher active memory ratio may not necessarily result in higher performance as shown in the figure due to the re-computation of victim instances.

Similarly, for lattice as shown in Fig. 9b, the baseline is still the worst. However, for *MCB*, it is competitive to, or even better than *DAR* in terms of the active memory. We know that *DAR* is more conservative than *MCB* in safety checking. Thus, this observation is inconsistent with those made in fork&join, which demonstrates again that the impact of safety checking on the workload's overall performance is diverse, mostly depending on the shape of the workflow graph. In this particular case, even though *DAR* could reserve more memory resources for the admitted instances, the average DOCs of each instance is not that high as those in fork&join case to improve the makespan performance (Fig. 7).

Since compared to other algorithms, *MCB* has low maximum claims, it thus can work with less memory resources with degraded performance, which is an advantage over others that allows *MCB* to carve the bits and pieces of instance admission out of the safety checking.

Finally, we simply compare the performance of *MCB* and Lang's algorithm for pipeline workloads (*DAR* is reduced to Lang's when acting on pipeline workloads).

In the experiment, we fix the memory capacity as 200 units and vary the number of pipeline stages from 4 to 22. Fig. 10 shows the results where in all cases, *MCB* outperforms Lang's algorithm, especially, as the number of stages increases.

It is easy to understand that with the increments of the stages, admission of new instances in Lang's algorithm is not affected too much because it is only determined by the maximum memory that can be used by a single task. On the contrary, in the same case, *MCB* can increase the caching memory utilization by reserving more memory for active instances. We also validated the explanation by comparing the changes of memory utilization of both algorithms as the stages increase, and observe that the relative ratios of three memory classes remain largely unchanged in Lang's algorithm, while in our case, the ratio of active (inactive) memory gradually increases (decreases), with the growth of the number of stages.

In summary, with these experiments, *MCB* exhibits advantages over other compared algorithms in the following aspects, which validates the values of MMC in memory-constrained workflow computing.

1) For fork&join with high average DOCs, *MCB* has a competitive performance with other compared algorithms by maximizing the average DOCs.
2) For lattice with low average DOCs, *MCB* has a better overall performance than others by ensuring high average DOCs, especially, the memory resources are increasing over a certain value.
3) For pipeline with low average DOCs, *MCB* consistently outperform Lang's algorithm across different stages.
4) For all cases, *MCB* can work with less memory resources with degraded performance, which provides the algorithm with a flexibility to control the workload execution.

7 Conclusion

In this paper we proposed a concept of *MinMax Memory Claim* that assures computational workflows to maximize the concurrency with minimum memory resources. We developed algorithms to compute it by reducing the problem to finding a maximum weighted clique in a derived graph via some graph transformation techniques and investigated efficient optimal solutions to a class of representative workflow graphs. With these results, we further designed a deadlock resolution algorithm that leverages the concept of MMC to improve the banker's algorithm by localizing the maximum claim of each instance in workloads at run-time.

To show the advantages of the concept of MMC, we implemented the proposed algorithms and applied them, through a simulation study, to deadlock avoidance in workflow-based workloads when in-memory caching are constrained in the cloud. Our results show that the proposed algorithms not only dominate the compared algorithms with respect to deadlock resolution but also exhibit some advantages over them to potentially improve the overall workload performance.

Acknowledgments

This work was supported in part by the China National Basic Research Program (973 Program, No. 2015CB352400), National Science Foundation of China under Grant No. 61672513, No. 61572377, U1401258 and 61550110250, Science and Technology Planning Project of Guangdong Province (2015B010129011, 2016A030313183), the Open Fund from SKLSE under Grant No. 2015-A-06, the Natural Science Foundation of Hubei Province of China under Grant No. 2014CFB239, and the Open Fund from HPCL under Grant No. 201512-02. Yang Wang is the corresponding author.

REFERENCES

[1] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *Proc. 3rd Workshop Workflows Support Large-Scale Sci.*, Nov. 2008, pp. 1–10.

[2] C. Hoffa, et al., "On the use of cloud computing for scientific workflows," in *Proc. 4th IEEE Int. Conf. eScience*, Dec. 2008, pp. 640–645.

[3] G. Juve, et al., "Data sharing options for scientific workflows on amazon EC2," in *Proc. ACM/IEEE Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2010, pp. 1–9.

[4] S. He, Y. Wang, and X.-H. Sun, "Boosting parallel file system performance via heterogeneity-aware selective data layout," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, pp. 2492–2505, Sep. 2016.

[5] S. He, X.-H. Sun, and Y. Wang, "Improving performance of parallel I/O systems through selective and layout-aware SSD cache," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 10, pp. 2940–2952, Oct. 2016.

[6] G. Chockler, G. Laden, and Y. Vigfusson, "Data caching as a cloud service," in *Proc. 4th Int. Workshop Large Scale Distrib. Syst. Middleware*, 2010, pp. 18–21.

[7] G. Chockler, G. Laden, and Y. Vigfusson, "Design and implementation of caching services in the cloud," *IBM J. Res. Develop.*, vol. 55, no. 6, pp. 9:1–9:11, Nov. 2011.

[8] R. M. Karp, *Reducibility Among Combinatorial Problems*, R. E. Miller and J. W. Thatcher, Eds. New York, NY, USA: Plenum, 1972.

[9] GROMACS, Apr. 2013. [Online]. Available: http://www.gromacs.org

[10] D. Szafron, et al., "Proteome analyst: Custom predictions with explanations in a Web-based tool for high-throughput proteome annotations," *Nucleic Acids Res.*, vol. 32, pp. W365–W371, 2004.

[11] T. Glatard, J. Montagnat, and X. Pennec, "Grid-enabled workflows for data intensive medical applications," in *Proc. 18th IEEE Symp. Comput.-Based Med. Syst.*, 2005, pp. 537–542.

[12] G. Cooperman, X. Ma, and V. H. Nguyen, "Static performance evaluation for memory-bound computing: The MBRAM model," in *Proc. Int. Conf. Parallel Distrib. Process. Techn. Appl.*, 2004, pp. 435–441.

[13] M. Abadi, M. Burrows, M. Manasse, and T. Wobber, "Moderately hard, memory-bound functions," *ACM Trans. Internet Technol.*, vol. 5, no. 2, pp. 299–327, May 2005.

[14] L. Watkins, R. Beyah, and C. Corbett, "Using network traffic to passively detect under utilized resources in high performance cluster grid computing environments," in *Proc. 1st Int. Conf. Netw. Grid Appl.*, 2007, pp. 16:1–16:8.

[15] J. D. Davis and E. S. Chung, "SpMV: A memory-bound application on the GPU stuck between a rock and a hard place," Microsoft Res., Redmond, WA, USA, Tech. Rep. MSR-TR-2012–95, Sep. 2012.

[16] A. Tiwari, A. Gamst, M. Laurenzano, M. Schulz, and L. Carrington, "Modeling the impact of reduced memory bandwidth on HPC applications," in *Euro-Par*, F. Silva, I. Dutra, and V. Santos Costa, Eds. Berlin, Germany: Springer, 2014, pp. 63–74.

[17] A. ElastiCache. (2016). [Online]. Available: http://aws.amazon.com/elasticache/

[18] Q. Wei, B. Veeravalli, B. Gong, L. Zeng, and D. Feng, "CDRM: A cost-effective dynamic replication management scheme for cloud storage cluster," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2010, pp. 188–196.

[19] M. Zhu, Q. Wu, and Y. Zhao, "A cost-effective scheduling algorithm for scientific workflows in clouds," in *Proc. IEEE 31st Int. Performance Comput. Commun. Conf.*, Dec. 2012, pp. 256–265.

[20] B. Palanisamy, A. Singh, and L. Liu, "Cost-effective resource provisioning for MapReduce in a cloud," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 5, pp. 1265–1279, May 2015.

[21] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 6:1–6:15.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–10.

[23] T. Rotaru, M. Rahn, and F.-J. Pfreundt, "MapReduce in GPI-space," in *Euro-Par 2013: Parallel Processing Workshops*, Berlin, Germany: Springer, 2014, pp. 43–52.

[24] D. Chiu, A. Shetty, and G. Agrawal, "Elastic cloud caches for accelerating service-oriented computations," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, Nov. 2010, pp. 1–11.

[25] T. Minoura, "Deadlock avoidance revisited," *J. ACM*, vol. 29, no. 4, pp. 1023–1048, Oct. 1982.

[26] S.-D. Lang, "An extended banker's algorithm for deadlock avoidance," *IEEE Trans. Softw. Eng.*, vol. 25, no. 3, pp. 428–432, May/Jun. 1999.

[27] Y. Wang and P. Lu, "Maximizing active storage resources with deadlock avoidance in workflow-based computations," *IEEE Trans. Comput.*, vol. 62, no. 11, pp. 2210–2223, Nov. 2013.

[28] S. Reveliotis and M. A. Lawley, "Efficient implementations of banker's algorithm for deadlock avoidance in flexible manufacturing systems," in *Proc. 6th Int. Conf. Emerg. Technol. Factory Autom.*, Mar. 1997, pp. 214–220.

[29] D. R. Wood, "An algorithm for finding a maximum clique in a graph," *Operations Res. Lett.*, vol. 21, no. 5, pp. 211–217, 1997.

[30] P. R. Östergård, "A fast algorithm for the maximum clique problem," *Discrete Appl. Math.*, vol. 120, no. 1, pp. 197–207, 2002.

[31] K. Ocaa, D. de Oliveira, F. Horta, J. Dias, E. Ogasawara, and M. Mattoso, "Exploring molecular evolution reconstruction using a parallel cloud based scientific workflow," in *Advances in Bioinformatics and Computational Biology*, M. de Souto and M. Kann, Eds. Berlin, Germany: Springer, 2012, pp. 179–191.

[32] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *J. Mol. Biol.*, vol. 215, no. 3, pp. 403–410, 1990.

[33] A. Rosenberg, "On scheduling mesh-structured computations for internet-based computing," *IEEE Trans. Comput.*, vol. 53, no. 9, pp. 1176–1186, Sep. 2004.

[34] P. Crandall, R. Aydt, A. Chien, and D. Reed, "Input/output characteristics of scalable parallel applications," in *Proc. IEEE/ACM Conf. Supercomputing*, 1995, pp. 59–89.

[35] P. Hulith, "The AMANDA experiment," in *Proc. 17th Int. Conf. Neutrino Phy. Astrophysics*, Jun. 1996, pp. 518–523.

[36] A. Sum and J. de Pablo, "Nautilus: Molecular simulation code," Dept. Chemical Eng., Univ. Wisconsin-Madison, Madison, WI, USA, 2002.

[37] B. Gendron, A. Hertz, and P. St-Louis, "A sequential elimination algorithm for computing bounds on the clique number of a graph," *Discrete Optimization*, vol. 5, no. 3, pp. 615–628, Aug. 2008.

[38] P. Gburzynski, "SMURPH," (2016). [Online]. Available: http://www.olsonet.com/pg/PAPERS/side.pdf

[39] Y. Wang and P. Lu, "DDS: A deadlock detection-based scheduling algorithm for workflow computations in HPC systems with storage constraints," *Parallel Comput.*, vol. 39, no. 8, pp. 291–305, Aug. 2013.

Shuibing He received the PhD degree in computer science and technology from Huazhong University of Science and Technology, China, in 2009. He is now an associate professor in the Computer School, Wuhan University, China. His current research areas include parallel I/O system, file and storage system, high-performance computing, and distributed computing.

Yang Wang received the BSc degree in applied mathematics from Ocean University of China, in 1989, the MSc degree in computer science from Carleton University, in 2001, and the PhD degree in computer science from the University of Alberta, Canada, in 2008. He is currently in Shenzhen Institute of Advanced Technology, Chinese Academy of Science, as a professor. His research interests include cloud computing, big data analytics, and Java virtual machine on multicores. He is an Alberta Industry R&D associate (2009-2011), and a Canadian Fulbright fellow (2014-2015).

Xian-He Sun received the BS degree in mathematics from Beijing Normal University, China, in 1982, and the MS and PhD degrees in computer science from Michigan State University, in 1987 and 1990, respectively. He is a distinguished professor in the Department of Computer Science, Illinois Institute of Technology (IIT), Chicago. He is the director of the Scalable Computing Software Laboratory, IIT, and is a guest faculty in the Mathematics and Computer Science Division, Argonne National Laboratory. His research interests include parallel and distributed processing, memory and I/O systems, software systems, and performance evaluation and optimization. He is a fellow of the IEEE.

Chengzhong Xu received the PhD degree from the University of Hong Kong, in 1993. He is currently a tenured professor with Wayne State University and the director of the Institute of Advanced Computing and Data Engineering, Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences. His research interests include parallel and distributed systems and cloud computing. He has published more than 200 papers in journals and conferences. He was the Best Paper Nominee of 2013 IEEE High Performance Computer Architecture (HPCA), and the Best Paper Nominee of 2013 ACM High Performance Distributed Computing (HPDC). He serves on a number of journal editorial boards, including the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Cloud Computing*, the *Journal of Parallel and Distributed Computing*, and the *Science China Information Sciences*. He received the Faculty Research Award, Career Development Chair Award, and the Presidents Award for Excellence in Teaching of WSU. He also received the Outstanding Oversea Scholar award of NSFC. He is a fellow of the IEEE.