An integrated approach to workflow mapping and task scheduling for delay minimization in distributed environments

Daqing Yun\textsuperscript{a}, Chase Qishi Wu\textsuperscript{b,}\textsuperscript{*}, Yi Gu\textsuperscript{c}

\textsuperscript{a} Department of Computer Science, University of Memphis, Memphis, TN 38152, United States
\textsuperscript{b} Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, United States
\textsuperscript{c} Department of Computer Science, Middle Tennessee State University, Murfreesboro, TN 37132, United States

HIGHLIGHTS

- We study workflow execution dynamics in distributed environments.
- We formulate an optimization problem on workflow mapping and task scheduling.
- We propose an integrated solution to maximize workflow performance.
- The proposed solution is evaluated through simulations and experiments.

ARTICLE INFO

Article history:
Received 13 February 2015
Received in revised form 18 June 2015
Accepted 7 July 2015
Available online 17 July 2015

Keywords:
Scientific workflows
Workflow mapping
On-node scheduling
End-to-end delay

ABSTRACT

Many scientific applications feature large-scale workflows consisting of computing modules that must be strategically deployed and executed in distributed environments. The end-to-end performance of such scientific workflows depends on both the mapping scheme that determines module assignment, and the scheduling policy that determines resource allocation if multiple modules are mapped to the same node. These two aspects of workflow optimization are traditionally treated as two separated topics, and the interactions between them have not been fully explored by any existing efforts. As the scale of scientific workflows and the complexity of network environments rapidly increase, each individual aspect of performance optimization alone can only meet with limited success. We conduct an in-depth investigation into workflow execution dynamics in distributed environments and formulate a generic problem that considers both workflow mapping and task scheduling to minimize the end-to-end delay of workflows. We propose an integrated solution, referred to as Mapping and Scheduling Interaction (MSI), to improve the workflow performance. The efficacy of MSI is illustrated by both extensive simulations and proof-of-concept experiments using real-life scientific workflows for climate modeling on a PC cluster.

© 2015 Elsevier Inc. All rights reserved.

1. Introduction

The processing and analysis of simulation or experimental datasets generated in next-generation e-Science require the construction and execution of domain-specific workflows in distributed network environments, such as clusters, grids, or clouds for collaborative research and discovery [23,33]. Such scientific workflows typically consist of many interdependent computing modules\textsuperscript{1}, and are managed and executed by either special- or general-purpose workflow engines such as HTCondor/DAGMan [18,26,20], Kepler [35], Pegasus [42,19], Triana [16], Askalon [24] and Sedna [45]. In general, a workflow system first assigns component modules to a set of networked nodes (i.e. mapping\textsuperscript{2}) in the deployment phase and then decides the order or priority of module execution (i.e. scheduling) at runtime.

The workflow mapping problem is well known to be NP-complete and non-approximable for planar and bipartite workflow graphs [25], and has been extensively investigated in the literature to minimize the End-to-end Delay (ED) or makespan of a work-
flow. No matter which mapping scheme is applied, it is often inevitable to assign multiple modules to the same node (i.e., node reuse) for better utilization of limited computing resources, leading to possible concurrent module execution. In such cases, the node's computing resources are generally allocated by kernel-level CPU scheduling policies, such as the round-robin algorithm to ensure fine-grained fair-share (FS). Similarly, a network link's bandwidth is also fairly shared by multiple independent data transfers that take place concurrently over the same link through the use of the widely deployed TCP or TCP-friendly transport methods. Such system-inherent fair-share scheduling mechanisms could reduce the development and implementation efforts of workflow systems, but may not always yield the best workflow performance, especially in distributed environments. The scheduling effect was considered in some recent efforts including [52], where a Critical Path-based Priority Scheduling (CPPS) algorithm is proposed to improve the end-to-end performance of a workflow under a given mapping scheme that is initially calculated based on FS scheduling.

As well recognized, the performance of scientific workflows depends on both the mapping scheme and the on-node scheduling policy, which are traditionally treated in two separate realms. As the scale of scientific workflows and the complexity of network environments rapidly increase, each individual aspect of performance optimization alone can only meet with limited success. In fact, there exists a certain level of interactions between mapping and scheduling, which could be exploited to further improve the end-to-end workflow performance. We shall use a simplified numerical example on a workflow consisting of 6 modules as shown in Fig. 1 to illustrate how mapping and scheduling can interact with each other, leading to different end-to-end performances. For simplicity, we ignore the computational requirement (CR) of the start/end module (i.e., $w_0$ and $w_5$). The initial mapping scheme maps modules $w_1$ and $w_2$ to node $v_1$, and maps modules $w_3$ and $w_4$ to node $v_2$. All the nodes have identical processing power (PP) of 10 units/s.

Under the given mapping scheme in Fig. 1, the execution dynamics of the entire workflow are shown in Fig. 2 based on FS scheduling. Along the x-axis, starting from time point 0, it takes 2 s of fair share between $w_1$ and $w_2$ on $v_1$ to finish $w_2$ and then another 3 s of exclusive execution of $w_1$ to finish $w_1$ at time point 5. At this point, $w_3$ and $w_4$ become "ready." It takes 2 s of fair share between $w_3$ and $w_4$ on $v_2$ to finish $w_4$ and another 1 s of exclusive execution of $w_3$ to finish $w_3$ at time point 8. Therefore, the end-to-end delay based on FS scheduling is 8 s on the critical path: $w_0 \rightarrow w_1 \rightarrow w_3 \rightarrow w_5$. Based on this FS schedule, by employing a naive on-node scheduling policy that always executes the critical module exclusively first in case of resource competition, we can cut down the end-to-end delay from 8 to 7 s as shown in Fig. 2. However, this new schedule leads to a different critical path, i.e., $w_0 \rightarrow w_2 \rightarrow w_4 \rightarrow w_5$. Such a shift of the critical path indicates that the original mapping scheme computed by any critical path-based mapping algorithm based on FS scheduling may no longer be the optimal one, hence requiring reexamining the mapping procedure.

Modern systems and networks provide various mechanisms to perform resource scheduling that goes beyond the system-inherent FS scheduling. For example, the allocation of system resources (mainly CPU cycles) on a host among concurrent modules or jobs could be controlled by assigning different CPU quantum values through "nicer" or "renice" commands or some specialized tools such as "CGroups" [12] in Linux. Meanwhile, apart from TCP-based data transfer over default best-effort IP paths, the allocation of network resources (mainly bandwidths) could also be controlled on dedicated channels, as exemplified by high-performance networks with the capability of bandwidth reservation [41,32] and the Internet using QoS technologies such as DiffServ [5], IntServ [8], RSVP [9], and MPLS [2] to implement fine-grained bandwidth control. Therefore, it is practically feasible to perform more sophisticated control for module execution and data transfer than the default FS scheduling to achieve a higher level of workflow performance.

In this paper, we conduct an in-depth investigation into workflow execution dynamics and formulate a generic workflow optimization problem that considers both workflow mapping and scheduling. We propose an integrated solution, referred to as Mapping and Scheduling Interaction (MSI), to optimize workflow performance. MSI takes an iterative approach to perform the mapping-scheduling cycle in order to minimize the end-to-end delay of a distributed workflow. The performance superiority of the proposed solution is illustrated by extensive simulations and proof-of-concept experiments using domain-specific scientific workflows in real network environments.

The rest of this paper is organized as follows. Section 2 conducts a survey of related work. Section 3 formulates the workflow optimization problem. Section 4 details the design of MSI algorithm. Section 5 evaluates the algorithm performance through simulations and experiments. Section 6 concludes our work.

2. Related work

We conduct a background survey on workflow optimization, which has been the focus of research in recent years [21,52,53,15]. Generally, there are two aspects of optimizing distributed tasks to improve scientific workflow performance: (i) assigning the component modules in a workflow to suitable computing resources, referred to as workflow mapping; and (ii) deciding the execution order/priority and resource sharing policy among concurrent modules on computer nodes or processors, referred to as workflow on-node scheduling. Both aspects have been extensively studied in various contexts due to their theoretical significance and practical importance [11,36,14,13,44,38,43,3,10,37].

The existing workflow mapping algorithms can be roughly classified into the following categories: (i) Graph-based methods [40,17], which tackle the mapping problem using graph theory. The subgraph isomorphism has been proven to be NP-complete and the complexity of graph isomorphism is still unknown; (ii) List scheduling approaches [30,34], most of which employ a critical path-based procedure; (iii) Clustering algorithms [7,28], which assume an unlimited number of processors and thus are not feasible in reality; (iv) Duplication-based algorithms [1,43], most of which have a complexity of $O(n^2)$ or higher for $n$ nodes; (v) Guided random search such as genetic algorithms [22,31], which often require additional efforts to determine an appropriate termination condition and usually does not have performance guarantee.

In early years when networked resources were still scarce, workflow modules were typically mapped to tightly coupled homogeneous systems such as multiprocessors [25,5,12,34]. As distributed platforms such as clusters, grids, and clouds are rapidly
developed and deployed, research efforts have shifted to workflow mapping in heterogeneous environments to utilize distributed resources at different geographical locations across wide-area networks, which facilitate a much larger scope of collaborations among different institutes [43,7,4]. However, some of the aforementioned efforts assume that networks are fully connected [44] or only consider independent tasks in the workflow [10] without any resource sharing. These assumptions are valid under certain circumstances, but may not be always practical nowadays and thus are difficult to be incorporated in real-life workflow applications.

Considering the limit in both the scope and the amount of networked resources, multiple modules are often mapped to the same node, which necessitates the scheduling of concurrent modules. Task scheduling has been a traditional topic in computer science, and fair-share scheduling is widely adopted in real systems. Some traditional scheduling algorithms consider multiple independent processes running on a single processor with a goal to optimize the waiting time, response time, and turnaround time of individual processes, or the throughput of the entire system. Garey et al. compiled a great collection of similar or related multi-processor scheduling problems in [27], most of which are NP-complete and were tackled by various heuristics such as list scheduling and simple level algorithm [29]. However, the research on module scheduling specifically tailored to scientific workflows in distributed environments is still quite limited. Many existing efforts on workflow scheduling assume non-pre-emption of module execution [37,10], which oversimplifies the execution dynamics in resource sharing and may not sufficiently model the complexity of real-life applications in wide-area networks.

The work in [52] investigates the task scheduling problem for Minimum End-to-end Delay (MED) in unitary processing applications under a given workflow mapping scheme, and shows that the performance could be improved by deciding an appropriate resource sharing policy among concurrent workflows mapped to the same node. In this paper, we go beyond the traditional mapping and scheduling approaches by integrating both to achieve a higher level of workflow performance.

3. Problem formulation

In this section, we construct analytical cost models and formally define a generic workflow optimization problem that takes both mapping and scheduling into consideration.

3.1. Cost models

We model a scientific workflow as a Directed Acyclic Graph (DAG) \( G_m(V_m, E_m), |V_m| = m \), where vertices represent computing modules starting from the start module \( w_0 \) to the end module \( w_{m-1} \). A directed edge \( e(i,j) \in E_m \) in the DAG represents the dependency between a pair of adjacent modules \( w_i \) and \( w_j \). As shown in Fig. 3(a), module \( w_i \) receives a data input of size \( z(i) \) from each of its preceding modules \( w_k, \ldots, w_j \) and performs a pre-defined computational routine. Generally, the computational requirement (CR) or workload of a module depends on the module’s computational complexity and input data size: the former is decided by the specific algorithm implemented in the module and the latter defines the problem size or the amount of data to be processed. We model the CR of a module \( m_i \) as a generic function \( f_i \) of the aggregated input data \( Z \):

\[
CR_i = f_i(Z_i) = f_i \left( \sum_{x=0}^{A} \left( z(x, i) \right) \right), \quad (1)
\]

where \( Z_i \) is an aggregation of all input data sizes based on a certain aggregation function \( A \). To make the computational complexity function generic, we do not specify the data aggregation method in our model, but make it concrete in practice. For example, in our simulations, we simply perform a summation operation on all the input data sizes for aggregation, i.e. \( A = \sum \).

In this workflow model, we consider a module as an atomic execution unit and ignore inter-module communication cost on the same node. For a workflow with multiple start/end modules, we can always convert it to this model by adding a virtual start/end module with zero computational requirement and connecting it to all the original start/end modules with zero data transfer.

We model an overlay computer network as an arbitrary weighted graph \( G_c(V_c, E_c), \) consisting of \( |V_c| = n \) nodes interconnected by \( |E_c| \) overlay links. The estimation of a computer node’s processing power (PP) is a complex task since it involves a variety of host factors. For simplicity, in Fig. 3(b), we use a normalized variable PP to represent the overall processing power of node \( v_i \). The communication link \( l(i,j) \) from \( v_i \) to \( v_j \) is associated with a certain bandwidth \( b(i,j) \) and minimum link delay (MLD) \( d(i,j) \). Generally, the start module \( w_0 \) serves as a data source on the source node \( v_0 \) without any computation to supply all initial data needed by the application and the end module \( w_{m-1} \) performs a terminal task on the destination node \( v_{n-1} \) without any further data transfer. This network model is generic to accommodate various distributed environments such as clouds, grids, and PC clusters.

The module’s complexity together with the input data size decides the amount of CPU cycles needed to finish the execution of the computational routine defined in the module. The actual execution time of a module also depends on the capacity of the processing power of the mapped computer node as well as its availability at runtime. Our model estimates the execution time of
Table 1
Notations in the cost models.

<table>
<thead>
<tr>
<th>Notations</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_a(V_a, E_a)$</td>
<td>A workflow, $</td>
</tr>
<tr>
<td>$w_i$</td>
<td>The $i$th module in $G_a$</td>
</tr>
<tr>
<td>$t^c_i$</td>
<td>The start time of module $w_i$</td>
</tr>
<tr>
<td>$t^d_i$</td>
<td>The finish time of module $w_i$</td>
</tr>
<tr>
<td>$e(i, j)$</td>
<td>The edge from module $w_i$ to $w_j$</td>
</tr>
<tr>
<td>$z(i, j)$</td>
<td>The data size sent from $w_i$ to $w_j$</td>
</tr>
<tr>
<td>$C_R$</td>
<td>The workload of module $w_i$</td>
</tr>
<tr>
<td>$G_v(V_v, E_v)$</td>
<td>An overlay network, $</td>
</tr>
<tr>
<td>$v_i$</td>
<td>The $i$th computing node</td>
</tr>
<tr>
<td>$P^c_i$</td>
<td>The processing power of node $v_i$</td>
</tr>
<tr>
<td>$l(i, j)$</td>
<td>The link from node $v_i$ to $v_j$</td>
</tr>
<tr>
<td>$b(i, j)$</td>
<td>The bandwidth of link $l(i, j)$</td>
</tr>
<tr>
<td>$d(i, j)$</td>
<td>The minimum link delay of link $l(i, j)$</td>
</tr>
<tr>
<td>$T_{trans}(z, l(i, j))$</td>
<td>The transfer time of data size $z$ over link $l(i, j)$</td>
</tr>
<tr>
<td>$T_{exec}(w, v)$</td>
<td>The execution time of module $w$ on node $v$</td>
</tr>
<tr>
<td>$M_{C_a} \rightarrow G_c$</td>
<td>An mapping scheme that maps $G_a$ to $G_c$</td>
</tr>
<tr>
<td>$M_i$</td>
<td>The mapping scheme in the $i$th round interaction of Algorithm 1</td>
</tr>
<tr>
<td>$S_{r(w_1, w_2, ..., w_k) \rightarrow v}$</td>
<td>An on-node scheduling policy for ${w_1, w_2, ..., w_k}$ over node $v$</td>
</tr>
<tr>
<td>$S_{a \rightarrow G_c}$</td>
<td>An entire on-node scheduling policy for $G_a$ on $G_c$</td>
</tr>
<tr>
<td>$T_i$</td>
<td>The $i$th interaction on-node scheduling policy in Algorithm 1</td>
</tr>
<tr>
<td>$T_{exec}(c_i, G_c)$</td>
<td>The execution time components of $G_a$ in the $i$th round in Algorithm 1</td>
</tr>
<tr>
<td>$T_{exec}(G_a, G_c, M, S)$</td>
<td>The execution time components of $G_a$ when mapped to $G_c$ by $M_{C_a} \rightarrow G_c$ and scheduled by $S_{a \rightarrow G_c}$</td>
</tr>
<tr>
<td>$CP(T)$</td>
<td>The critical path calculated based on time components $T$</td>
</tr>
<tr>
<td>$ED(G_a, G_c, M, S)$</td>
<td>End-to-end delay of $G_a$ when mapped to $G_c$ by $M_{C_a} \rightarrow G_c$ and scheduled by $S_{a \rightarrow G_c}$</td>
</tr>
</tbody>
</table>

The workflow module $w_i$ on computer node $v_j$ as follows:

$$T_{exec}(w_i, v_j) = \frac{C_R}{PP} = \frac{f(i(Z))}{PP}.$$  \hspace{1cm} (2)

The transfer time of a large chunk of data over a communication link is mainly constrained by the link’s bandwidth while the MLD can be a significant overhead for the transfer of a small amount of data whose size is comparable to the Maximum Transmission Unit (MTU) of the underlying network infrastructure. We estimate the transfer time of a message of data size $z$ over a communication link $l(i, j)$ as follows:

$$T_{trans}(z, l(i, j)) = \frac{z}{b(i, j)} + d(i, j).$$ \hspace{1cm} (3)

Based on the above models, we may use an existing workflow mapping algorithm to compute a mapping scheme under the following constraints on workflow mapping and execution/transfer precedence [52]:

- Each module/edge is required to be mapped to only one node/link.
- A computing module cannot start execution until all its required input data arrives.
- A dependency edge cannot start data transfer until its preceding module finishes.

We define a workflow mapping scheme $M_{C_a} \rightarrow G_c$ that maps $G_a$ to $G_c$ as follows:

**Definition 1** (Workflow Mapping Scheme).

$$M_{C_a} \rightarrow G_c = \left\{ \begin{array}{l}
\forall v \in V_a, \exists v \in V_c; \\
l(v_i, v_j) \in E_c, \\
\text{if } \langle w_p \rightarrow v_i, w_q \rightarrow v_j, e(w_p, w_q) \rangle \in E_a, \\
\quad 0 \leq p, q \leq |V_a| - 1, \quad 0 \leq i, j \leq |V_c| - 1.
\end{array} \right.$$ \hspace{1cm} (4)

The requirements for a workflow mapping scheme defined in Definition 1 are two-fold: (i) for any module $w$ in the workflow $G_a$, there must be a node $v$ in the computer network $G_c$, to which $w$ is mapped ($w \rightarrow v$); (ii) if modules $w_p$ and $w_q$ are mapped to nodes $v_i$ (i.e. $w_p \rightarrow v_i$) and $v_j$ (i.e. $w_q \rightarrow v_j$), respectively, and there is an edge $e(w_p, w_q)$ between $w_p$ and $w_q$ in the workflow $G_a$, there must be a link $l(v_i, v_j)$ between $v_i$ and $v_j$ in the network $G_c$.

We further convert the above workflow and network models to virtual graphs as follows: each dependency edge in the workflow is replaced with a virtual module whose computational requirement is equal to the corresponding data size, and each mapped network link is replaced by a virtual node whose processing power is equal to the corresponding bandwidth. This conversion facilitates workflow scheduling as we only need to focus on the module execution time.

When there are multiple modules (either actual workflow module or converted virtual module) mapped to one single computer node, the CPU time (quantum) allocated to each competing module is controlled by the kernel-level CPU scheduling policy, and the link bandwidth allocated to each data transfer is determined by the specific transport methods used on the end hosts. In the previous work [50,48,49,30], both CPU time and link bandwidth are assumed to be fairly shared through the use of a round-robin scheduling algorithm and a TCP or TCP-friendly transport protocol. In this work, we go beyond this assumption and use an *On-node Scheduling Policy* to determine resource sharing in a more fine-grained manner.

**Definition 2** (On-node Scheduling Policy). If a subset of modules $\{w_1, w_2, ..., w_k\}$ in a workflow $G_a$ are mapped to a computer node $v$ and there exists resource competition among these modules, an On-node Scheduling Policy for $\{w_1, w_2, ..., w_k\}$ on $v$, denoted as $S_{w_1, w_2, ..., w_k \rightarrow v}$, is a vector in which each element $s_i$ $(1 \leq i \leq k)$ is the resource share quantum for $w_i$ in a scheduling round.

Since an on-node scheduling policy is node-specific according to Definition 2, we use $S_{a \rightarrow G_c}$ to denote a full set of on-node scheduling policies for the entire workflow $G_a$ over computer network $G_c$. Once $M_{C_a} \rightarrow G_c$ and $S_{a \rightarrow G_c}$ are both determined for all modules and nodes, we could calculate the execution time of every workflow module and the End-to-end Delay (ED) of the entire workflow, as defined below.

**Definition 3** (End-to-end Delay (ED)). The End-to-end Delay (ED) of a workflow $G_a(V_a, E_a)$ that is mapped to a computer network $G_c$ by $M_{C_a} \rightarrow G_c$ and scheduled by $S_{a \rightarrow G_c}$, denoted as $ED(G_a, G_c, M, S)$, is the time duration from the time point when the first module $w_0$ in $G_a$ starts execution to the time point when the last module $w_{|V_a| - 1}$ in $G_a$ is finished.

We tabulate the notations used in the cost models in Table 1 for convenience, some of which are used in the algorithm design in Section 4.

3.2. Workflow optimization problem

We formally define a generic *End-to-end Workflow Optimization Problem* (EWOP) that considers both mapping and scheduling as follows.

**Definition 4** (End-to-end Workflow Optimization Problem (EWOP)). Given a DAG-structured computing workflow $G_a(V_a, E_a)$, and a heterogeneous overlay computer network $G_c(V_c, E_c)$, we wish to find a mapping scheme $M_{C_a} \rightarrow G_c$ together with an on-node scheduling policy $S_{a \rightarrow G_c}$ that achieves the minimum $ED(G_a, G_c, M, S)$.

The general workflow mapping problem is well known to be NP-complete, and so is the on-node scheduling problem [52]. As a combined optimization problem, EWOP with preemptive scheduling is also NP-complete [27].

We would like to point out that for workflow execution, different stakeholders have different interests, which may not be always
aligned well with each other. From the perspective of end users such as research scientists, the main goal is to execute the workflow as fast as possible to improve the research productivity and efficiency in their domains; while from the perspective of resource owners such as cloud service providers, the main goal is to efficiently utilize (limited) computing resources to satisfy the needs of different computing tasks from different users with different requirements. Furthermore, commercial service providers oftentimes have another goal to maximize their financial profit, which may also be in conflict with the other objectives.

In this paper, our goal is to minimize the workflow execution time and help scientists improve the research efficiency in accomplishing their scientific missions. In a typical context for scientific computing, the computing resource is either owned by the institution where the end user is employed or pre-allocated before the actual execution of the workflow. Under these circumstances, the other concerns such as resource cost or utilization may not be as critical as the mission of the research.

4. Algorithm design

We design a Mapping and Scheduling Interaction (MSI) algorithm to solve EWOP, which optimizes workflow end-to-end delay by exploring the interactions between workflow mapping and on-node scheduling. We use a top-down approach to present the MSI algorithm: first describe the overall algorithm structure and then provide design details for each component algorithm.

Algorithm 1 MSI(Gw, Gc)
Input: a workflow Gw(Vw, Ew), an overlay network Gc(Vc, Ec).
Output: a mapping scheme Mw\rightarrow Gc, an on-node scheduling policy Scw\rightarrow Gc, and the resulted Minimum End-to-end Delay (MED).
1: \textbf{if} i \leftarrow 1; T1 \leftarrow \emptyset; MED \leftarrow \infty;
2: \textbf{while} the termination condition is not met \textbf{do}
3: \quad M_w, T_{(G_w, G_c, M, G_f)} \leftarrow CPMM(G_w, G_c, T);
4: \quad ED_{(G_w, G_c, M, G_f)} \leftarrow CP(T_{(G_w, G_c, M, G_f)});
5: \quad S_i, T_{(G_w, G_c, M, G_f)} \leftarrow EEDC(G_w, G_c, M_i, T_{(G_w, G_c, M, G_f)});
6: \quad ED_{(G_w, G_c, M, G_f)} \leftarrow CP(T_{(G_w, G_c, M, G_f)});
7: \quad MED \leftarrow \min \{ED_{(G_w, G_c, M, G_f)}; ED_{(G_w, G_c, M, G_f)}; MED\}, \text{ and update } T_i, M_{(G_w, G_c, M, G_f)} \text{ and } S_{(G_w, G_c, M, G_f)}; \text{ associated with } MED;
8: \textbf{if} i \leftarrow i + 1;
9: \textbf{return} MED, M_{(G_w, G_c, M, G_f)} \text{ and } S_{(G_w, G_c, M, G_f)};

4.1. Mapping and scheduling interaction algorithm

As shown in Algorithm 1, the MSI algorithm conducts an optimization process in a repetitive manner. In Line 3, MSI first calculates an initial mapping scheme that maps all modules to the network using the Critical Path-based Module Mapping (CPMM) algorithm (Algorithm 2 in Section 4.2.1), which in turn uses the Efficient Linear Pipeline Configuration (ELPC) algorithm [50] (Section 4.2.1) for critical module mapping and the Greedy Branch Module Mapping (GBMM) algorithm (Algorithm 3 in Section 4.2.2) for branch module mapping. CPMM yields a mapping scheme M_i and calculates the time cost components T_i(G_w, G_c, M_i, G_f), based on which, MSI computes the critical path CP(T_i(G_w, G_c, M_i, G_f)) and the corresponding end-to-end delay ED_i(G_w, G_c, M_i, G_f) in Line 4. In Line 5, MSI determines an on-node job scheduling policy S_i using the Exact End-to-end Delay Calculation (EEDC) algorithm (Algorithm 4 in Section 4.3), which invokes the Critical Path-based Priority Scheduling (CPPS) algorithm (Algorithm 5 in Section 4.4) to perform on-node scheduling on each Concurrent Module Set (CMS), and then calculates the execution time components T_i(G_w, G_c, M_i, G_f) based on the current schedule S_i. After calculating the end-to-end delay in Line 6, MSI determines the current best (minimum) End-to-end Delay (MED) obtained by the current best mapping scheme M_{Gw\rightarrow Gc} and on-node scheduling policy S_{Gw\rightarrow Gc} and calculates the corresponding time cost components T_i for the next round of interaction (Lines 7 and 8). MSI repeats the above process until a certain termination condition is met, for example, when it reaches a pre-specified number of iterations or the performance improvement between two adjacent iterations does not exceed a preset threshold. Particularly, in our implementation, we employ a simple but effective method to control the termination condition of MSI, which involves two constant parameters: I_{imp} and I_{max} that denotes the number of contiguous interactions where no improvement is observed over the preceding one and I_{max} that denotes the maximum number of interactions for a guaranteed termination. The algorithm terminates whichever constant is reached first. These two parameters are customizable and could be adjusted for different applications.

A high-level flowchart of the MSI algorithm is provided in Fig. 4.

Algorithm 2 CPMM(Gw, Gc, T)
Input: a workflow Gw(Vw, Ew), an overlay network graph Gc(Vc, Ec), and time cost components T of Gw.
Output: a mapping scheme M_{Gw\rightarrow Gc}, and corresponding time cost components T(G_w, G_c, M, G_f).
1: \textbf{if} T \leftarrow \emptyset \textbf{then}
2: \quad Assume that Gc is complete with homogeneous nodes and links;
3: \quad Calculate the initial execution time components T_{init} of each module;
4: \quad T \leftarrow T_{init};
5: \quad Calculate the critical path CP(T) based on T;
6: \quad Map the critical modules on the critical path CP(T) using ELPC for MED-ANR, and update M_{Gw\rightarrow Gc};
7: \quad Map branch modules using GBMM in Algorithm 3, and update M_{Gw\rightarrow Gc};
8: \quad Calculate time components T_{(G_w, G_c, M, G_f)} based on M_{Gw\rightarrow Gc} and FS using extED[30];
9: \quad return M_{Gw\rightarrow Gc} and T_{(G_w, G_c, M, G_f)};

4.2. Critical path-based module mapping algorithm

The Critical Path-based Module Mapping (CPMM) algorithm first maps all the modules on the critical path (i.e. critical modules) to the network using the ELPC algorithm in [49] (Line 6 in Algorithm 1), and then uses a greedy approach, Greedy Branch Module Mapping (GBMM) (Line 7 in Algorithm 2) to decide the mapping scheme for the workflow modules that are not on the critical path (i.e. branch modules). We describe these two algorithms in Section 4.2.1 and Section 4.2.2, respectively.

4.2.1. ELPC algorithm for critical module mapping

The critical module mapping problem is formally defined as a Minimum End-to-end Delay with Arbitrary Node Reuse (MED-ANR) problem in [49]. Efficient Linear Pipeline Configuration (ELPC) in [49] are a set of algorithms for pipeline mapping in different problem scenarios, among which the optimal one solves MED-ANR problem using a dynamic programming (DP) approach. MSI incorporates ELPC for MED-ANR to determine the mapping scheme for critical modules (Line 6 in Algorithm 2). Note that in the initial phase, CPMM assumes that the network is fully-connected with
homogeneous computer nodes to calculate the initial time cost components of each module as well as the initial critical path and the corresponding end-to-end delay, as shown from Line 1 to Line 5 in Algorithm 2. After the first round of interaction, CPMM removes this assumption and uses the updated time cost components for later interactions.

Algorithm 3 GBMM(Gw, Gc, CP)
Input: A workflow Gw(Vw, Ew), an overlay network Gc(Vc, Ec) and a mapped critical path CP.
Output: A mapping scheme for all branch modules in Gw, denoted as M'.
1: Insert mapped modules on CP into a queue Q with a sequential order from start module w0 to end module w|Vw|−1;
2: while Q = \\{\} do
3:    wj ← Dequeue(Q);
4:    Prioritize all unmapped direct succeeding modules of wj according to workloads, denoted as pri(suc(wj));
5:    for all wj ∈ pri(suc(wj)) from high to low do
6:       f(wj) ← g(wj, wj) + h(wj, w|Vw|−1) using A* and Beam Search;
7:       Select the node nj that produce minimal f(wj) to map wj;
8:       Update M';
9:    Enqueue(Q, wj);
10:   return M';

4.2.2. Greedy branch module mapping algorithm

Once the mapping scheme for critical modules is determined, CPMM uses Greedy Branch Module Mapping (GBMM) algorithm to compute the mapping scheme for branch modules (see Line 7 in Algorithm 2). The pseudocode of GBMM is shown in Algorithm 3, which uses a sorting-based breath first search approach to compute the mapping scheme for branch modules. To improve the naive greedy approach, GBMM employs a cost function f(wj) = g(wj, wj) + h(wj, w|Vw|−1) for module wj ∈ G(Ew, Vw) in A* and Beam Search, which contains two parts: (i) a path cost function g(wj, wj), calculated from the source node to the current node, which may or may not be optimal; and (ii) a heuristic estimation function h(wj, w|Vw|−1) of the cost from the current node to the destination node. The greedy procedure of GBMM in the calculation of h(wj, w|Vw|−1) in Algorithm 3 always maps the current module to the computer node that produces the minimum execution time at the current stage without considering whether this would affect the future mapping or the end-to-end delay. To reduce the complexity of the search algorithm, we also explore a predetermined parameter \( \lambda \) as the least-cost path in the search tree of the solution space, instead of searching all possible paths.

4.3. Exact End-to-end delay calculation algorithm

We design an on-node Exact End-to-end Delay Calculation (EEDC) algorithm to perform on-node scheduling and calculate module execution time components in MSI. The EEDC algorithm is a modified version of the extED algorithm in [30] with an integrated on-node scheduling procedure CPSS shown in Algorithm 5. It produces a set of processor scheduling policies for all mapping nodes that collectively cut down the execution time of the critical path in the workflow to achieve MED.

The pseudocode of the EEDC algorithm is provided in Algorithm 4. Under the given mapping scheme \( M_{Gw→Gc} \) (calculated by Algorithm 2), it first calculates an independent set (see Line 3 in Algorithm 4 for definition) of each module (from Line 6 to Line 8), and then finds all modules with earliest start time, as set0 (see Line 11). If there is only one module in set0, this module is executed immediately to its completion; otherwise, an on-node scheduling strategy over all the concurrent modules in set0 needs to be decided. In Line 16, we find the concurrent module set (CMS) set, with the earliest start time. The CMS of a module \( w \), denoted as CMS(w), is a set that contains all the modules that are mapped to the same node as \( w \), and executed concurrently with resource contention. According to Algorithm 5, we recalculate the share quantum of the processing power \( per(w) \) that is allocated to each module \( w \) in set1 in Line 17. Note that a complete on-node scheduling policy is comprised of all on-node scheduling policies of all the concurrent module sets on their corresponding mapping nodes, which are determined in each while-loop (Line 10). We use “++” in Line 17 to denote the “update” operation so that at the end, a complete on-node scheduling policy \( S_{Gw→Gc} \) for all the modules on their corresponding mapping nodes is returned. Once the on-node scheduling policy is decided, this new on-node scheduling policy is applied to all concurrent modules for their executions and time cost calculations (from Line 18 to Line 30).

Algorithm 4 EEDC(Gw, Gc, \( M_{Gw→Gc}, T_{Gw,Gc,M,FS} \))
Input: a converted workflow Gw(Vw, Ew), a converted overlay network Gc(Vc, Ec), a mapping scheme Mw→Gc, and the corresponding execution time components T_{Gw,Gc,M,FS} of Gw.
Output: an on-node scheduling policy \( S_{Gw→Gc} \) and corresponding time cost components T_{Gw,Gc,M,S}.
1: t'(set): set of start times of all modules in a set;
2: t'(set): set of finish times of all modules in a set;
3: ids(w): independent module (i.e. modules among which no dependence exists) set of module \( w \) mapped to the same node (excluding \( w \));
4: est(set) = \{w | w ∈ set and t'(set) is ‘ready’ to execute);
5: for all wj ∈ Vw do
6:    Find ids(wj);
7:    Set wj as ‘unfinished’;
8:    Set wj as ‘ready’;
9:    while there are ‘unfinished’ modules in Vw do
10:       Find set0 = est(ready(Vw));
11:      for all module wj ∈ set0 do
12:         if \( \{ids(wj)\} = \{\} \) then
13:            Finish wj, calculate T\( \\langle exec(wj)\rangle \), update T\( \langle Gw,Gc,M,FS\rangle \) with it, and set wj as ‘finished’;
14:            else
15:               Find set1 = {w | w is ‘ready’ and w ∈ est(ids(wj) ∪ \( w \))};
16:               S ← CPPS; Gw, Gc, M, T\( \langle Gw,Gc,M,FS\rangle \), \( T_{Gw,Gc,M,S}\); set1;
17:               Estimate t'(set1);
18:               Find set2 = \{w | w ∈ ids(wj) ∪ \( w \) s.t. t'(w) < min(t'(set1), and w is ‘ready’ and ‘unfinished’);
19:               if \( \{set2\} > 0 \) then
20:                  for all module wj ∈ set1 do
21:                     From t1 = t'(w)(w ∈ set1) to t2 = min(t'(set2)), execute wj under set; 2) calculate partial of T\( \\langle exec(wj)\rangle \), and update \( T_{Gw,Gc,M,S}\) :
22:                     Update the new t'_w = min(t'(set2));
23:                  end
24:               else
25:                  for all module wj ∈ set1 do
26:                     if t'_w = min(t'(set1)) then
27:                        Finish wj, calculate T\( \\langle exec(wj)\rangle \), update T\( \langle Gw,Gc,M,FS\rangle \) with it, and set wj as ‘finished’;
28:                        else
29:                           From t1 = t'(w)(w ∈ set1) to t2 = min(t'(set1)): execute wj under set; 2) calculate partial of T\( \\langle exec(wj)\rangle \), and update \( T_{Gw,Gc,M,S}\) :
30:                           Update the new t'_w = min(t'(set1));
31:                          Mark all ready modules as ‘ready’;
32:                       return S and \( T_{Gw,Gc,M,S}\) ;
4.4. Critical path-based priority scheduling algorithm

We design a Critical Path-based Priority Scheduling (CPPS) algorithm in Algorithm 5 to perform the actual on-node scheduling for each CMS. The module execution in EWOP is preemptive and hence leads to complex resource sharing dynamics. We first provide an in-depth analysis of resource sharing.

4.4.1. Analysis of resource sharing dynamics

The end-to-end delay of a workflow is determined by its critical path. In general, to minimize the end-to-end delay, we need to allocate more resources to critical modules than those branch modules when they are in the same CMS. In EWOP, the module execution is preemptive, which results in more complex resource sharing dynamics. We present three lemmas on the resource sharing dynamics among the concurrent modules assigned to the same node, i.e., within a CMS, which provide us the guidelines in the design of our scheduling algorithm.

Lemma 1. The minimal finish time of a CMS consisting of \( k \) concurrent modules executed on a single fully-utilized processor with processing power \( PP \) is a constant, \( \sum_{i=1}^{k} CR_i / PP \).

Proof. Since the total workload of all \( k \) modules is fixed, i.e. \( \sum_{i=1}^{k} CR_i \), and the processor is fully operating, no matter how the modules are scheduled, the total execution time would remain unchanged and is always bounded by the finish time of the last completed module.

To better understand the significance of on-node scheduling during workflow execution, we investigate a typical scheduling scenario shown in Fig. 5 where the best performance improvement could be achieved over a fair-share scheduling policy under the following conditions:

- There are \( k \) modules running concurrently on the same node \( v \) during their entire execution time periods;
- Among \( k \) modules, one is a critical module and the other \( k-1 \) are branch modules, and all of them are of the same workload;
- Each branch module is the only module on its execution path (except for the start and end modules).

Based on the scheduling scenario in Fig. 5, we have Lemma 2:

Lemma 2. The MED performance improvement of any workflow with a fixed critical path achievable by any on-node scheduling over fair-share scheduling is upper bounded by 50%.

Proof. We first justify that the conditions in Fig. 5 are necessary for achieving the upper bound of the workflow’s MED improvement.

Let \( x \) be the original execution time of the critical module on node \( v \) using the fair-share scheduling policy. If there were any non-critical modules on \( v \) that finished before \( x \), even if we let the critical module \( w \) run exclusively first, it would take longer than \( x/k \) to complete. Thus, we would not be able to reduce the execution time of the critical module to the minimum time possible, i.e. \( x/k \). On the other hand, if there were any non-critical modules on \( v \) that finished after \( x \), from Lemma 1, we know that the total execution time on this node would be greater than \( x \), and hence we would not be able to reduce the length of the critical path to \( x \).

The above discussion concludes that all \( k \) modules on node \( v \) must share resources during their entire execution time and finish at the same time to achieve the maximum possible reduction on the length of the critical path. Since we consider fair-share as the comparison base in Lemma 1, the CRs of all \( k \) modules must also be identical. In this scheduling scenario, if we let the critical module \( w \) execute exclusively first, its finish time \( t_{w}^{f} \) would be reduced from \( x \) to \( x/k \), which is the largest possible improvement with \( k \) concurrent tasks on the same node.

We use \( y \) to denote the sum of the execution time for the rest of the modules on the critical path (excluding \( w \)). From Lemma 1, we have \( \max(t_{w}^{f}) = x \), \( i = 2, 3, \ldots, k \), which means that the latest finish time of the last completed branch module would be still \( x \). Since the new length of the critical path (i.e. MED) becomes \( (x/k) + y \), the upper bound MED improvement is achieved if this new length is equal to the latest finish time of any branch module, i.e. \( (x/k) + y = x \). It follows that \( y = (k-1)/k \times x \). Therefore, the MED improvement over fair share is \( \Delta = (x+y)/x = (k-1)/(2k-1) \), which is 50% as \( k \to \infty \).

We define several key terms, which are used in Lemma 3.

Definition 5. The execution time of a module \( w \) is the time duration starting from the time point when \( w \) becomes “ready” to the time point when \( w \) finishes its execution completely.

In our model, \( w \) is executed immediately once it becomes “ready” unless the scheduling policy suspends it and allocates computing resources to other concurrent modules. According to Definition 5, \( w \)’s execution time includes its waiting time.

Definition 6. The Global Critical Path (GCP) of a DAG-structured workflow \( G_{w}(V_{w}, E_{w}) \) is the longest path of execution time from the start module \( w_{0} \) to the end module \( w_{|V_{w}|} \).

Definition 7. The Nodal Critical Path (NCP) of module \( w \) in a DAG-structured workflow is the longest path of execution time traversing module \( w \).

Based on Lemmas 1 and 2, we have the following lemma to illustrate the condition, under which an optimal scheduling solution is achievable.

Lemma 3. The MED of a DAG-structured workflow is achieved if and only if when the Nodal Critical Paths of all workflow modules that share resources with the critical modules are scheduled to have the same length as the Global Critical Path.

Proof. Without loss of generality, we consider a workflow where a concurrent module set \( \{w_{1}, w_{2}, \ldots, w_{k}\} \) is mapped to the same node \( v \), as shown in Fig. 6. Suppose that the nodal critical path \( NCP(w_{1}) \) is the global critical path of the workflow. Branch modules \( w_{2}, \ldots, w_{k} \) are mapped to the same node \( v \) with resource competition. There exist \( k \) NCPs that go through \( v \), denoted as \( NCP_{1}, NCP_{2}, \ldots, NCP_{k} \), with their lengths denoted as \( l_{1}, l_{2}, \ldots, l_{k} \).
respectively (assume that \(l_1 \geq l_2 \geq \cdots \geq l_k\)). If they are not of equal length, since NCP\(_i\) is the longest, one can always allocate more resource of \(v\) to the critical module \(w_1\) such that \(l_1\) would become shorter and the other NCPs would become longer.

If we move resources to the critical module from those branch modules in some reasonable manner, theoretically, we should be able to obtain a new mapping and scheduling strategy where all NCPs are equal with a new length \(l_{eq}\), where \(l_1 \leq l_{eq} \leq l_k\). The same is also true for other critical modules that share resources with branch modules.

4.4.2. Design of CPPS

The key idea of CPPS is to “squeeze” an appropriate amount of computing resources out of branch modules and re-allocate them to critical modules. Towards this end, a critical task is to decide the percent \(\text{per}(w_i)\) of CPU allocation for each module \(w_i\) in the concurrent module set CMS\((w_i)\).

CPPS schedules all concurrent module sets in a left-to-right manner based on the execution time components under fair-share as well as the partial execution time components under the new on-node scheduling policy. In Algorithm 5, from Line 8 to Line 10, it first checks all the modules in the CMS: if there is no critical module, a fair-share schedule is simply returned (from Line 11 to Line 13); otherwise, CPPS attempts to let the critical modules execute with more processing power reallocated from those branch modules in CMS. If such a new on-node scheduling does not change the original critical path of the workflow, CPPS returns this schedule; otherwise, it estimates the maximal amount of processing power that can be moved from the branch modules to the critical modules, as shown from Line 14 to Line 34.

To estimate the amount of processing power for reallocation, CPPS first estimates the execution times \(T_{\text{Ex}}\) of critical modules if they are executed exclusively (from Line 14 to Line 15). Since multiple NCPs may have the same length, if there are multiple critical modules, they share the processing power in a fair manner.

In Line 16, the latest finish time \(\text{lf}(v)\) of each module mapped onto node \(v\) is calculated according to Lemma 1. For each branch module \(w_1\) (Line 18), we estimate the time \(\text{T}_{\text{exec}}(\text{tb}(w_1))\) it needs to finish (Line 19), and the amount of time that can be possibly delayed, denoted as \(\delta t_1\) and \(\delta t_2\) (Line 20 and Line 21). We extend the execution time of the branch module by \(\delta t_1\) or \(\delta t_2\) (Line 24), whichever has a lower possibility to change the original critical path. If this extension does not change the original critical path based on the estimation indicated by \(\beta\) in Line 22, the share quantum \(\text{per}(w_i)\) of processing power allocated to each module \(w_i\) is calculated based on \(\Delta t\) (from Line 23 to Line 26). Otherwise, we mark it using a boolean variable \(\text{flag}\) (Line 28).

To further explain how \(\delta t_1\) and \(\delta t_2\) are calculated, we consider a CMS with three modules \(w_1, w_2\) and \(w_3\) that are mapped to a computer node \(v\). These three modules start executing at the same time, and module \(w_3\) is the critical module, which is the last one to be finished. Fig. 7(a) shows the execution dynamics of this CMS under fair-share scheduling. If we let \(w_3\) execute exclusively ahead of \(w_1\) and \(w_2\), the execution dynamics are shown in Fig. 7(b), where dotted lines represent possible start and finish time points of \(w_1\) and \(w_2\), which depend on the workloads of \(w_1\) and \(w_2\), the processing power of \(v\), and the specific on-node scheduling policy on the CMS. No matter how their executions are scheduled, either \(w_1\) or \(w_2\), or both of them would finish at the latest possible finish time \(\text{T}_{\text{exec}}(\text{tb}(v))\) according to Lemma 1. Therefore, for each branch module \(w_i \in \{w_1, w_2\}\), in general, we have the following inequality:

\[
\text{T}_{\text{exec}}(\text{tb}(w_i)) + \delta t_1 \leq \text{lf}(v),
\]

where \(\delta t_1\) is the amount of execution time that can be delayed for the branch module.

Algorithm 5 CPPS\((G_w, G_c, M_w \rightarrow G_c, T_{(G_w, G_c \rightarrow \text{CMS})}, T_{(G_w, G_c \rightarrow \text{CMS})})\)

**Input:** a converted workflow \(G_w(V_w, E_w)\), a converted overlay network \(G_c(V_c, E_c)\), a given mapping scheme \(M_{w \rightarrow G_c}\), time cost components \(T_{(G_w, G_c \rightarrow \text{CMS})}\) of all modules under fair-share, execution time components \(T_{(G_w, G_c \rightarrow \text{CMS})}\) of partially scheduled modules under CPPS, and an concurrent modules set CMS

**Output:** On-node scheduling policy \(\text{S}_{\text{CMS} \rightarrow v}\) for all modules in CMS on computer node \(v\).

1. \(v\): computer node to which modules in CMS are mapped;
2. \(\text{LNCP}(w_i)\): the left segment of NCP\(_i\) (from \(w_1\) to \(w_2\));
3. \(\text{RNCP}(w_i)\): the right segment of NCP\(_i\) (from \(w_1\) to \(w_{w_i-1}\));
4. \(\text{per}(w_i)\): percent of processing power allocated to module \(w_i\) from \(v\);
5. \(\text{lf}(v)\): the latest possible finish time of \(w_i\) in CMS node \(v\);
6. \(T_{\text{Ex}}(w_i)\): execution time of \(w_i\) when it is executed exclusively;
7. \(\text{tb}(w_i)\): the partial workload of module \(w_i\) to be finished;
8. for all \(w_i \in \text{CMS}\) do
9. \(\text{NCP}(w_i) \leftarrow \text{LNCP}(w_i) + \text{RNCP}(w_i);\)
10. Find \(\text{set}_0\) \(\leftarrow \{w|\forall w, \text{NCP}(w) \leq \text{CP}(T_{\text{Ex}}(w))\};\)
11. if \(\text{set}_0\) \(\Rightarrow 0\) then
12. \(S_{\text{CMS} \rightarrow v} \leftarrow \text{FS};\)
13. return \(S_{\text{CMS} \rightarrow v};\)
14. for all \(w_i \in \text{set}_0\) do
15. Estimate execution time \(T_{\text{Ex}}(w_i));\)
16. Calculate \(\text{lf}(v);\)
17. \(\text{bool flag} \leftarrow \text{false};\)
18. for all \(w_i \in \text{CMS} \& w_i \notin \text{set}_0\) do
19. Estimate \(T_{\text{exec}}(\text{tb}(w_i));\)
20. \(\delta t_1 \leftarrow \min(\text{lf}(v) - T_{\text{exec}}(\text{tb}(w_i)), T_{\text{Ex}}(\{w_i | w_i \in \text{set}_0\});\)
21. \(\delta t_2 \leftarrow \text{CP}(T_{(G_w, G_c \rightarrow \text{CMS})}) - \text{NCP}(w_i);\)
22. \(\beta \leftarrow \left\{\text{\text{NCP}(w_i)} \leq \text{CP}(T_{(G_w, G_c \rightarrow \text{CMS})})\left\};\)
23. if \(\text{NCP}(w_i) + \beta \times \min(\delta t_1, \delta t_2) \leq \text{CP}(T_{(G_w, G_c \rightarrow \text{CMS})})\) then
24. \(\Delta t \leftarrow T_{\text{exec}}(\text{tb}(w_i)) + \min(\delta t_1, \delta t_2);\)
25. Calculate \(\text{per}(w_i)\) based on \(\Delta t;\)
26. Mark \(w_i;\)
27. else
28. \(\text{flag} \leftarrow \text{true};\)
29. if \(\text{flag} \Rightarrow \text{true}\) then
30. for all \(w_i \in \text{CMS}\) but is not marked do
31. \(\text{per}(w_i) \leftarrow \left(1 - \frac{\sum_{w_i \in \text{CMS} \& w_i \text{ is marked}}}{\text{per}(w_i)}\right) \left|\left| w_i \text{ is not marked}\right|\right|;\)
32. else
33. Let all \(w_i \in \text{set}_0\) exclusively fair-share \(v\)'s processing power and suspend all other modules in \(\text{set}_0\), update \(\text{per}(\cdot);\)
34. Update \(S_{\text{CMS} \rightarrow v}\) with \(\text{per}(\cdot);\)
35. return \(S_{\text{CMS} \rightarrow v};\)

If \(w_1\) is scheduled for exclusive execution ahead of \(w_1\) and \(w_2\), we have

\[
\delta t_1 = T_{\text{Ex}}(w_3),
\]

\[
T_{\text{exec}}(\text{tb}(w_i)) + T_{\text{Ex}}(w_3) \leq \text{lf}(v) \quad (i = 1, 2).
\]

To avoid critical path shifting, which may lead to a longer end-to-end delay, the execution time of the branch module cannot be overly extended. Hence, we calculate \(\delta t_1\) as follows (Line 20):

\[
\delta t_1 = \min\left\{\text{lf}(v) - T_{\text{exec}}(\text{tb}(w_i)), T_{\text{Ex}}(w_3)\right\}.
\]

Also, any delay on the finish time of branch module \(w_i\) would also increase the length of NCP\(_i\). For \(w \in \{w_1, w_2\}\), we further avoid critical path shifting by imposing the following conditions to
When \( \delta_2 \) is needed; \( \delta_2 \leq \text{CP}(T_{(G_w, G_c, M, FS)}) \), \( \beta \) in Line 22 is the number of the common modules that the workflow’s GCP and all independent modules’ NCPs share. If this extension does not change the original critical path, we extend the amount of execution time for branch modules by \( \min(\delta_1, \delta_2) \), and then use flag to indicate whether there are some branch modules that may change the GCP (Line 28). If flag is changed to be true, we let all critical modules and those marked modules (Line 26) fairly share the node’s processing power with the amount reallocated from the branch modules (from Line 29 to Line 31); otherwise, since GCP is not changed by the new on-node scheduling policy, we let the critical module exclusively execute before the branch modules (from Line 32 to Line 33).

The on-node scheduling policy \( S_{CMS-\beta} \) is returned as a vector \( \text{per} \cdot () \) in Algorithm 5, where we consider two cases:

- When \( \text{per}(w|w \in \text{set}_B) = 1 \) and \( \text{per}(w|w \notin \text{set}_B) = 0 \): we exclusively execute the critical module until some module finishes (where a new on-node scheduling policy \( S' \) is needed); while for the other modules, we suspend them to wait for the critical module to finish. During this period of time, the critical module may be entirely or partially completed. We then execute all the modules (their unfinished part \( tbf(w) \)) that are mapped to node \( v \) based on the new on-node scheduling policy \( S' \) until the number of concurrent modules assigned to \( v \) changes again.

- When all \( \text{per} \cdot () \) are between 0 and 1: we execute each module with its own share quantum \( \text{per} \cdot () \) of resource allocation until the number of concurrent modules changes.

We provide a simple numerical example to illustrate the CPPS scheduling process. As shown in Fig. 8, a computing workflow \( G_w \) consisting of 9 modules \( w_0, w_1, \ldots, w_8 \) is mapped to a computer network \( G_c \) consisting of 5 nodes \( v_0, v_1, \ldots, v_4 \). The workloads of \( G_w \) and the mapping scheme for \( G_w \) are tabulated in Table 2. The processing powers of the computer nodes are \( PP_1 = 10 \text{ units/s}, PP_2 = 10 \text{ units/s} \), respectively. Under a fair-share (FS) scheduling policy, the start time and end time of each module are provided in Table 2, where \( G_w \) takes 20 s to complete along the critical path consisting of modules: \( w_0 \rightarrow w_2 \rightarrow w_4 \rightarrow w_5 \rightarrow w_7 \rightarrow w_8 \).

For CPPS, Fig. 9 illustrates the scheduling status when modules \( w_3 \) and \( w_4 \) are being scheduled by CPPS in Algorithm 5. At this point of time, the left part has been scheduled by CPPS while the right part is estimated based on FS (we tabulate the execution time in Table 2 and use parentheses to differentiate those that have been produced by CPPS). Note that the module’s execution start time is always calculated from the point when the module is “ready” to execute, not from the point when it actually starts running. Right after \( w_1 \) finishes execution, both \( w_3 \) and \( w_4 \) are “ready” to execute with potential resource competition. To decide the sharing policy between them, we need to compute the critical path that traverses each of them (NCP) by concatenating the left and right segments of its critical path. For \( w_3 \), the length of its left CP segment \( w_0 \rightarrow w_1 \rightarrow w_3 \), denoted as \( LNCP(w_3) \), is 5 under CPPS, while the length of its right CP segment \( w_3 \rightarrow w_7 \rightarrow w_8 \), denoted as \( RNCP(w_3) \), is 8 including \( w_3 \)'s own execution time, which is estimated based on FS. Similarly, for \( w_4 \), the lengths of its \( LNCP(w_4) \) and \( RNCP(w_4) \) are 5 under CPPS and 13 under FS, respectively. Since \( NCP(w_3) = LNCP(w_3) + RNCP(w_3) = 13 \) is longer than \( NCP(w_4) = LNCP(w_4) + RNCP(w_4) = 18 \), \( w_3 \) is to be executed with more computing resource. We estimate that the length of \( NCP(w_3) \) would not exceed the length of the original global critical path based on FS, which is 20, even if we let \( w_4 \) run to its completion with exclusive use of processing power. So, in this case, we assign the entire \( PP_1 \) to \( w_3 \) until it finishes; otherwise, \( w_4 \) would execute exclusively until a certain point and then share with \( w_3 \) in a fair-share manner such that the length of \( NCP(w_3) \) does not exceed the length of the original global critical path based on FS. This scheduling process is repeated on every computer node with multiple modules mapped until all the modules are properly scheduled.

### 4.5. Time complexity of MSI

According to Lines 2 and 5 in Algorithm 3, the time complexity of GBMM is \( O(\lambda \cdot m \cdot |E_w|) \), where \( \lambda \) is the predetermined number of least-cost paths to reduce the complexity, \( m \) is the number
of modules in the workflow, and $|E_w|$ is the number edges in the workflow. According to Lines 2–8 in Algorithm 2, the time complexity of CPMM is $O(\lambda \cdot m \cdot |E_w|) = O(\lambda \cdot m \cdot |E_w| + m^3) = O(\lambda \cdot m \cdot |E_w| + m^3)$ since the time complexity of ELPC and extED are $O(m \cdot |E_w|)$ and $O(m^3)$, respectively [49,30]. According to Lines 8, 14, and 18 in Algorithm 5, the time complexity of CPPS is $O(\alpha \cdot m)$ since running time complexity of calculating a critical path in a DAG is linear, where $\alpha$ is the number of modules in the CMS. According to Lines 10, 12, 17, 21 and 25 in Algorithm 4, the time complexity of EEDC is $O(\alpha \cdot m + m \cdot |E_w|) = O(m \cdot |E_w| + m^3)$. 

As such, according to Lines 2, 3, and 5 in Algorithm 1, the overall time complexity of MSI is $O(k \cdot (\lambda \cdot m + \alpha \cdot m^2 + m)) = O(k \cdot (\lambda + \alpha) \cdot n \cdot |E_w|)$, where $k$ is the number of iterations, $\lambda$ is a predefined number of least-cost paths, $\alpha$ is the number of modules in the CMS that depends on the workflow structure and specific mapping and scheduling schemes, and $n$ is the size of the workflow (i.e., the number of modules).

5. Performance evaluation

We evaluate the performance of the proposed MSI algorithm in both simulated and experimental environments in comparison with various existing methods.

5.1. Simulations

We implement the MSI algorithm in C++ and run it on a Windows 7 desktop PC equipped with Inter(R) Core(TM)2 Duo CPU E7500 of 2.93 GHz and 4.00 GB memory.

We adopt a similar program in [30] to generate test datasets with variousexistingmethods. We evaluate the performance of the proposed MSI algorithm in both simulated and experimental environments in comparison with various existing methods.

5.1. Simulations

We implement the MSI algorithm in C++ and run it on a Windows 7 desktop PC equipped with Inter(R) Core(TM)2 Duo CPU E7500 of 2.93 GHz and 4.00 GB memory.

We adopt a similar program in [30] to generate test datasets with variousexistingmethods.

5.1.1. Simulation results

To evaluate the performance of MSI, we randomly generate 4 groups of test cases with 4 different numbers of nodes, i.e. $n = 5, 10, 15, 20$. In each group, we generate test cases in 20 different problem sizes, indexed from 1 to 20, by varying the number of modules from 30 to 125, at an interval of 5 with a random number of edges.

For each problem size ($m, |E_w|, n$), we generate 20 random problem instances for workflow optimization with different module complexities, data sizes, node processing powers, and link bandwidths, which follow a uniform distribution in an appropriate range of values. We collect the performance measurements for Simple Greedy, Greedy A* [39], impRCP [30], and MSI, and calculate the average of 20 instances in each case. The end-to-end delay comparison between these algorithms is plotted in Fig. 10. The MED performance improvement or speedup of MSI over impRCP, defined as $\frac{\text{MED}_{\text{impRCP}} - \text{MED}_{\text{MSI}}}{\text{MED}_{\text{impRCP}}} \times 100\%$, is tabulated in Table 3 and further plotted in Figs. 10 and 11 with both the mean and standard deviation for a visual examination.

Table 2

<table>
<thead>
<tr>
<th>Module</th>
<th>$w_0$</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>$w_4$</th>
<th>$w_5$</th>
<th>$w_6$</th>
<th>$w_7$</th>
<th>$w_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload (units)</td>
<td>0</td>
<td>10</td>
<td>40</td>
<td>20</td>
<td>10</td>
<td>80</td>
<td>60</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>Mapped to</td>
<td>$v_1$</td>
<td>$v_1$</td>
<td>$v_1$</td>
<td>$v_1$</td>
<td>$v_1$</td>
<td>$v_2$</td>
<td>$v_3$</td>
<td>$v_4$</td>
<td>$v_4$</td>
</tr>
<tr>
<td>Start time (FS)</td>
<td>0 s</td>
<td>0 s</td>
<td>0 s</td>
<td>0 s</td>
<td>2 s</td>
<td>7 s</td>
<td>8 s</td>
<td>8 s</td>
<td>16 s</td>
</tr>
<tr>
<td>End time (FS)</td>
<td>0 s</td>
<td>2 s</td>
<td>7 s</td>
<td>6 s</td>
<td>8 s</td>
<td>16 s</td>
<td>14 s</td>
<td>20 s</td>
<td>20 s</td>
</tr>
<tr>
<td>Start time (CPPS)</td>
<td>(0 s)</td>
<td>(0 s)</td>
<td>(0 s)</td>
<td>(5 s)</td>
<td>(5 s)</td>
<td>(5 s)</td>
<td>(5 s)</td>
<td>(8 s)</td>
<td>(8 s)</td>
</tr>
<tr>
<td>End time (CPPS)</td>
<td>(0 s)</td>
<td>(5 s)</td>
<td>(4 s)</td>
<td>(8 s)</td>
<td>(6 s)</td>
<td>(16 s)</td>
<td>14 s</td>
<td>20 s</td>
<td>20 s</td>
</tr>
<tr>
<td>Start time (CPPS)</td>
<td>(0 s)</td>
<td>(5 s)</td>
<td>(0 s)</td>
<td>(5 s)</td>
<td>(5 s)</td>
<td>(5 s)</td>
<td>(6 s)</td>
<td>(6 s)</td>
<td>(14 s)</td>
</tr>
<tr>
<td>End time (CPPS)</td>
<td>(0 s)</td>
<td>(5 s)</td>
<td>(4 s)</td>
<td>(6 s)</td>
<td>(6 s)</td>
<td>(14 s)</td>
<td>(12 s)</td>
<td>(18 s)</td>
<td>(18 s)</td>
</tr>
</tbody>
</table>

Table 3

<table>
<thead>
<tr>
<th>Problem Module #</th>
<th>Average MED Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>$m$</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td>60</td>
</tr>
<tr>
<td>8</td>
<td>65</td>
</tr>
<tr>
<td>9</td>
<td>70</td>
</tr>
<tr>
<td>10</td>
<td>75</td>
</tr>
<tr>
<td>11</td>
<td>80</td>
</tr>
<tr>
<td>12</td>
<td>85</td>
</tr>
<tr>
<td>13</td>
<td>90</td>
</tr>
<tr>
<td>14</td>
<td>95</td>
</tr>
<tr>
<td>15</td>
<td>100</td>
</tr>
<tr>
<td>16</td>
<td>105</td>
</tr>
<tr>
<td>17</td>
<td>110</td>
</tr>
<tr>
<td>18</td>
<td>115</td>
</tr>
<tr>
<td>19</td>
<td>120</td>
</tr>
<tr>
<td>20</td>
<td>125</td>
</tr>
</tbody>
</table>

In our simulations, we observe that the performance improvement largely depends on the mapping scheme since it determines the resource sharing level during workflow execution. In small problem sizes, or when the number of modules is comparable to the number of nodes, the modules are likely to be mapped to the nodes in a uniform manner, resulting in a low level of resource sharing, unless there exist some nodes whose processing power is significantly higher than the others in the network. Hence, in these cases, the MED improvement of MSI over impRCP is not very obvious. However, as the problem size increases, more modules might be mapped to the same node with more resource sharing, leading to a higher performance improvement. This overall trend of performance improvement is reflected in Figs. 10 and 11.

The amount of resource sharing in the workflow execution depends on not only the number of modules, but also the structure of the workflow and the complexities of modules. For a workflow with a small number of modules in each layer, only a limited amount of resource sharing may occur. If the complexity of a critical module $w$ is significantly larger than that of others that compete for resource, running $w$ exclusively would only yield a marginal performance improvement. This overall trend of performance improvement is also reflected in Figs. 10 and 11.

The above comparison between MSI and impRCP shows that the performance improvement varies significantly from case to case because it highly depends on the level of resource sharing in the test case, as indicated by the standard deviations in Fig. 11. Generally, as long as such resource sharing exists with limited resources, MSI is able to achieve a consistently better end-to-end performance than the other methods.

5.1.2. MSI optimization evolution process

MSI alleviates the negative effect of the critical path shifting on workflow mapping caused by on-node scheduling and eventually
improves the overall end-to-end performance. Starting from an initial mapping scheme, MSI keeps searching for a relatively “better” critical path by exploring the interactions between mapping and scheduling to optimize the workflow performance.

To examine the microscopic behavior of MSI involving both mapping and scheduling, we look into one typical test case of problem index 14 with parameters \( m = 95, |E| = 420, n = 10 \). We plot the performance measurements at all the iterations in...
Fig. 12. The interaction convergence curve in Experiment 14 with parameters $m = 95$, $|E_w| = 420$, $n = 10$ in Fig. 10(b).

We observe that the end-to-end delay is continuously improved when such a shifting occurs until the algorithm converges to a stable point.

5.2. Proof-of-concept experiments

To evaluate the efficacy of the MSI algorithm, we apply it to a real-life scientific workflow, Weather Research and Forecasting (WRF) [46], provided by Brookhaven National Laboratory (BNL), as shown in Fig. 13. We compose such a workflow using WPS-WRF modules, and map and execute it in a local PC cluster, which contains 16 compute nodes equipped with 12 Intel(R) Xeon(R) CPUs of 1200 MHz and 64 G RAM.

5.2.1. Weather research and forecasting (WRF)

The Weather Research and Forecasting (WRF) model [46] from BNL [6] has been widely used for regional to continental scale weather forecast. The workflow for typical applications of WRF model takes multiple steps, including data preparation and preprocessing, actual model simulation, and postprocessing. Each step could be computationally intensive and/or involve a large amount of data transfer and processing. Moreover, because of the chaotic nature of the atmospheric system and unavoidable errors in the input data and the imperfection of the model, ensemble approaches have to be adopted with a sufficiently large number of simulations. A workflow-based management and execution is hence extremely useful to automate the procedure and efficiently allocate the resources to carry out the required computational tasks.

5.2.2. Composite climate modeling workflow structure

As shown in Fig. 14, we duplicate the entire WPS-WRF workflow to generate five parallel pipelines to process five different dataset instances of the workflow input. The sizes of 5 input datasets are: 423.38 MB (Pipeline 1), 106.15 MB (Pipeline 2), 106.15 MB (Pipeline 3), 106.15 MB (Pipeline 4), and 106.15 MB (Pipeline 5), respectively. By doing so, we are able to construct a relatively complex workflow that contains 32 modules. We employ the performance modeling and profiling approach in [47] to determine the workload of each module prior to the actual execution. We map the composite workflow to 10 compute nodes ($n_1$ to $n_{10}$) in the PC cluster to process five batches of climate modeling input datasets. Since $w_0$ and $w_{31}$ are virtual start and end modules, they are not mapped to any physical compute node. Note that each pipeline in Fig. 14 corresponds to the climate modeling workflow shown in Fig. 13, i.e. geogrid $\leftrightarrow w_1$, ungrib $\leftrightarrow w_2$, metgrid $\leftrightarrow w_3$, real $\leftrightarrow w_4$, wrf $\leftrightarrow w_5$, ARWpost $\leftrightarrow w_6$, and so on.

5.2.3. A PC cluster-based workflow engine

To automatically execute the entire composite workflow shown in Fig. 14, we develop a lightweight PC Cluster-based Workflow Engine, referred to as ClusterWE, to manage the workflow execution under the inter-module precedence constraints.

ClusterWE is a daemon program that runs on every compute node with at least one workflow module mapped. As shown in Fig. 15, ClusterWE has a controller that includes a sender for sending a finish (“FIN”) signal to the succeeding node(s) and a receiver for receiving a finish signal from the preceding node(s), and an executor for scheduling and executing workflow module(s). The controller runs on a different core from the executor to avoid computing resource competition with workflow modules (cores 1 and 2, as shown in Fig. 15). The workflow structure, workflow mapping scheme, and executable list are stored in three different configuration files. Fig. 16 shows the format and contents of the workflow structure file and mapping scheme file. ClusterWE manages the workflow execution as follows:

1. A ClusterWE instance runs on each compute node with module(s) mapped, and parses the workflow structure, mapping scheme and executable information from the configuration files.
2. The receiver of the controller (as a daemon thread) listens on a pre-defined port number for a one-byte finish (“FIN”) message that indicates the completion of a preceding module. The delay in transmitting this “FIN” message is considered negligible compared to the inter-module data transfer delay in the workflow.
3. Once all finish messages for one mapped module are received from its preceding modules, this module becomes “ready”
and starts running within the executor on another core as a newly created thread. When there exists computing resource competition among multiple modules executed simultaneously (either multiple mapped modules are “ready” or some modules are already executing within the executor), the executor employs an on-node scheduling policy, either FS or CPPS, to control the execution priorities of those concurrent modules.

4. Once a module finishes, the sender of the controller sends a finish message to each of its succeeding modules.

ClusterWE employs multi-threading to increase the performance measurement accuracy. Since each compute node in the PC cluster has 12 CPU cores, we set the CPU affinities of the controller and the executor to be any two most lightly loaded cores on the mapping node to minimize the negative effect of undesired resource competition and system dynamics for performance evaluation.

5.2.4 Experimental results

We run the workflow shown in Fig. 14 for 10 times under both FS as used by existing methods, and CPPS as used by MSI in the PC cluster, and calculate the average execution time of each workflow module, as tabulated in Table 4. The critical path of execution time under FS is: $w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow w_4 \rightarrow w_5 \rightarrow w_6 \rightarrow w_7$, and the corresponding end-to-end delay is 1987.12 s; while under MSI, the critical path is: $w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow w_4 \rightarrow w_5 \rightarrow w_6 \rightarrow w_7$, and the corresponding end-to-end delay is 1498.02 s. Compared with FS scheduling under a given mapping scheme, MSI achieves an average 24.61% performance improvement in terms of end-to-end delay by invoking a more sophisticated CPPS on-node scheduling procedure.

6. Conclusion

We proposed an integrated Mapping and Scheduling Interaction (MSI) algorithm that incorporates scheduling into mapping for workflow performance optimization. Extensive simulation and experimental results on real-life scientific workflows illustrated that MSI is able to improve the end-to-end performance by exploring the interactions between mapping and scheduling. It is of our future interest to perform this integration in a dynamic manner and test it on scientific workflows of larger scales in wide-area production networks. It is also of our future interest to explore finer-grained control mechanisms beyond system commands “nice” and “renice” to realize a higher precision in CPU allocation for further performance improvement of MSI.

Acknowledgment

This research is sponsored by Department of Energy’s Office of Science under Grant No. DE-SC0010641 with University of Memphis.

References


