Scheduling and Analysis of Real-Time OpenMP Task Systems with Tied Tasks

Jinghao Sun[1,2], Nan Guan[2], Yang Wang[1], Qingqing He[1] and Wang Yi[1,3]

[1]Northeastern University, China
[2]The Hong Kong Polytechnic University, Hong Kong
[3]Uppsala University, Sweden

***Abstract*—OpenMP is a promising programming framework to develop parallel real-time systems on multi-cores. Although similar to the DAG task model, the OpenMP task systems are significantly more difficult to analyze due to various constraints posed by the OpenMP specification. An important feature in OpenMP is the `tied` tasks, which must execute on the same thread during the whole life cycle. Although `tied` tasks enjoy benefits in simplicity and efficiency, it was considered to be not suitable to real-time systems due to its complex behavior. In this paper, we study the real-time scheduling and analysis of OpenMP task systems with `tied` tasks. First, we show that under the existing scheduling algorithms adopted by OpenMP, `tied` tasks indeed may lead to extremely bad timing behaviors where the workload of a parallel task system is sequentially executed. To solve this problem, we propose a new scheduling algorithm and we developed two response time bounds for the new algorithm, with different trade-off between simplicity and analysis precision. Experiments with both randomly generated OpenMP task systems and realistic OpenMP programs show that the response time bounds obtained by our new scheduling algorithm and analysis techniques for `tied` task systems are very close to that of `untied` tasks, which may also be a good choice for real-time systems in many cases.**

I. Introduction

Multi-cores are more and more widely used in real-time systems to meet the rapidly increasing requirements in high performance and low power consumption. To fully utilize the computation power of multi-core processors, software must be parallelized. OpenMP [1] is a popular parallel programming framework, which is not only widely used in general and high-performance computing, but also has drawn increasing interests in embedded and real-time computing [2]–[5].

A fundamental problem in real-time system design is the scheduling of the workload. A common way to model parallel software system is using Directed Acyclic Graph (DAG). OpenMP supports explicit task systems since version 4.0. The execution semantics of OpenMP task systems are closely related to the DAG task models, and this motivates many theoretical work on scheduling and analysis of DAG task models [5]–[12]. However, the OpenMP language framework poses many constraints to the workload model and runtime scheduling behavior, which cannot be fully captured by the DAG task model. Therefore, the results with DAG task models are often not applicable to realistic OpenMP task systems.

An important feature in OpenMP task systems is introduced by the `tied` keyword, which enforces a task (which includes several vertices in a DAG) to execute on the same thread during its life cycle. The `tied` tasks enjoy the following benefits [1], [4]: (1) `tied` task precludes migrations among threads, which simplifies the implementation of the scheduling algorithm and reduces context switch cost; (2) `tied` tasks automatically reduce deadlocks in the presence of critical sections. In OpenMP, all tasks are `tied` by default, unless an `untied` keyword is explicitly added when the task is created.

However, `tied` tasks and related constraints in OpenMP bring significant challenge to the real-time scheduling and analysis of OpenMP task systems. A recent work [4] showed that the scheduling algorithms adopted by OpenMP are work-conserving if all tasks are `untied`, to which the classical response time bound [13] for DAG task model is applicable. However, when the task system contains `tied` tasks, the scheduling algorithms in OpenMP are not work-conserving, and consequently "*a timing analysis for tied tasks, besides being conceptually very difficult to achieve, would require to address sources of inherent complexity that would lead to unacceptably pessimistic response-time bounds*" [4]. So far, the problem of scheduling OpenMP task systems containing `tied` tasks with guaranteed response time bounds is open.

In this paper, we address the above open problem. First, we show that under the existing algorithms adopted by OpenMP, `tied` tasks not only make the response time analysis difficult, but indeed may lead to extremely bad timing behavior: almost all the workload of a parallel task system is tied to a single thread, and thus has to be executed sequentially. Therefore, the existing algorithms in OpenMP are indeed not suitable to real-time systems where guaranteed worst-case response time bounds are required.

To solve this problem, we propose a new algorithm to schedule OpenMP task systems, which at runtime uses simple rules to avoid tying too much workload to the same thread. Then we developed two response time bounds for the new algorithm, with different trade-off between simplicity (efficiency) and analysis precision. We conduct experiments with both randomly generated OpenMP task systems and realistic OpenMP programs to evaluate our proposed scheduling algorithm and analysis techniques. Experiment results show that in most cases the response time bounds obtained by our new scheduling algorithm and analysis techniques for `tied` task systems are very close to that of `untied` tasks, which suggests that `tied` tasks can also be used in OpenMP-based real-time systems.

II. Related Work

Much work has been done on scheduling DAG-based parallel real-time task systems [5]–[7]. The task models in these papers are closely related to the workload model of OpenMP, but missing many features in realistic OpenMP programs. Recently, some of these features have been taken into consideration. Motivated by Task Scheduling Point (TSP) in OpenMP, some work has been done on the scheduling of parallel tasks with limited preemption points [14].

Recently, Vargas et al [3] and Serrano et al [4] studied the possibility to apply OpenMP to real-time systems mainly from the real-time scheduling perspective. These work highlighted some important features in OpenMP that are relevant to real-time scheduling. In particular, they discussed how the Task Scheduling Point (TSP) and Task Scheduling Constraints (TSC) affect the real-time scheduling behaviour.

Serrano et al [4] studied the problem of bounding the response time (makespan) of the (non-recurring) task system generated by an OpenMP application on multi-cores. [4] developed response time bounds for the case containing only `untied` tasks, and claimed that bounding the response time in the presence of `tied` tasks is inherently hard. However, in this paper we will show that for OpenMP programs that contains `tied`, the response times can also be well bounded as for the case having only `untied` tasks.

III. Preliminary

In this section, we introduce the concepts and notations related to the OpenMP task system and its runtime scheduling. For simplicity of presentation, we focus on a *single non-recurrent* OpenMP application. However, our results are also applicable to a system with multiple recurrent OpenMP applications (with different periods) using the federated scheduling framework, as discussed in a technical report [15].

A. An Overview of OpenMP Program

An OpenMP program starts with a `parallel` directive (e.g., Line 1 in List. 1), which constructs the associated `parallel` region including all code enclosed in the pair of brackets following the `parallel` directive (e.g., Lines 2 to 23 in List. 1).

1) OpenMP Threads: The `parallel` directive also creates a team of n OpenMP threads (n being specified with the `num_threads` clause). The OpenMP thread is an execution entity which is able to execute the computation within the `parallel` region. In the rest of the paper, the term "thread" refers an OpenMP thread, and similar to previous work, each thread is assumed to exclusively execute on a dedicated core.

Listing 1: An example OpenMP program

```
1  #pragma omp parallel num_threads (n) {
2  #pragma omp single { // τ1
3     part10; // P10
4     #pragma omp task { // τ2
5        part20; // P20
6        #pragma omp task { // τ3
7           part30; // P30
8           #pragma omp task depend(out:x){//τ4
9              part40; } // P40
10          part31; // P31
11          #pragma omp task depend( in:x){//τ5
12             part50; } // P50
13          part32; // P32
14          #pragma omp task depend(out:x){//τ6
15             part60; } // P60
16          part33; }// P33
17       part21; // P21
18       #pragma omp task { // τ7
19          part70; } // P70
20       part22;  // P22
21       #pragma omp taskwait;
22       part23; } // P23
23    part11; }}  // P11
```

2) OpenMP Tasks: The code in the `parallel` region has a parallelism structure, which consists of a set of independent parallel units, called OpenMP *tasks*. In the rest of the paper, the term *task* refers an OpenMP task. A task is either *implicit* or *explicit*. All explicit tasks are annotated by `task` directives (e.g., τ_2, Line 4 at List. 1). The *body* of a task includes the code that is closely enclosed in the pair of brackets following the `task` directive (e.g., the code in Lines 5, 6, 17, 18, 20, 21 and 22 belongs to task τ_2). On the other hand, the code that is not associated with any `task` directive belongs to an implicit task (e.g., code at Lines 3, 4 and 23 in List. 1 is contained in an implicit task τ_1). For simplicity, we focus on the explicit tasks annotated by `task` directives in this paper.

In the following we introduce some basic notations related to tasks.

Task Relations

For any task τ, its associated `task` directive is assumed to be closely enclosed in the body of a task τ'. In this case, we say τ is the *child* task of τ', and τ' is the *parent* task of τ. Two tasks share the same parent task are *siblings*.

Moreover, a task τ is the descendant of τ' if τ is the (grand)child of τ', and in this case, τ' is the ascendant of τ. A task τ is the *non-descendant* task of τ' if τ is not the descendant of τ'.

For example, in List. 1, the task τ_2 is the child of task τ_1. Tasks τ_4, τ_5 and τ_6 are siblings, since they share the common parent τ_3. Moreover, task τ_4 is a descendant of τ_1. Task τ_2 is an ascendant of τ_4.

Task Creation and Completion

A task τ is *created* when a thread is executing the parent task of τ and encounters the `task` construct of τ. A task is *completed* when its last code of its body is executed.

For example, in List. 1, τ_2 is created when its parent task τ_1 is executed (See in Line 4). Moreover, τ_2 is completed when the code at Line 23 is executed.

Task Synchronization

Tasks synchronize with each other via two different mechanisms below.

- `taskwait` clause. Parent task can synchronize with its children via `taskwait` clauses. When encountering `taskwait`, the parent task is blocked until all of its first level children created beforehand have been finished.
- `depend` clause. Sibling task synchronize with each other via `depend` clauses. The `depend` clause enforce an order among the sibling tasks. If a task has an `in` dependence on a variable, it cannot start execution until all its previously created siblings with `out` or `inout` dependences on the same variable have been completed. On the other hand, if a task has an `out` or `inout` on a variable, it cannot start execution until all its previously created siblings with `in`, `out` or `inout` dependences on the same variable have been completed.

For example, in List. 1, task τ_2 synchronizes with its children τ_3 and τ_7 via `taskwait` clause at Line 21. More specifically, a thread cannot execute τ_2 (the code at Line 22) unless τ_3 and τ_7 are completed. Moreover, the siblings τ_4, τ_5 and τ_6 synchronize with each other via `depend` clauses. More specifically, τ_5 waits for the completion of τ_4. τ_6 waits for the completion of τ_4 and τ_5.

3) Schedule and Scheduling Rules: Given a set of tasks and a team of threads, a *schedule* is an assignment of tasks to threads, so that each task is executed until completion. In OpenMP, a feasible schedule must fulfil the following three constraints.

Task Scheduling Points

The body of a task is sequentially executed. The execution of a task can only be interrupted at task scheduling points (TSP). TSP occurs upon task creation and completion, as well as at task synchronization points such as `taskwait` directives, explicit and implicit barriers[1]. For example, TSPs divide the program in List. 1 into several *parts* (e.g., `part10`, `part11`, ect.), and the TSP exists between any two adjacent parts. More specifically, the *parts* of an OpenMP program divided by TSPs are non-preemptive.

`tied` and `untied` Tasks

In OpenMP, a task is either `tied` or `untied`.

The `tied` task forces the code in its body to be executed on the same thread. More specifically, If a `tied` task starts execution on a thread, then it will only execute on this thread in its whole life cycle. In particular, if the execution of a `tied` task is interrupted, later it must be resumed on the same thread.

In contrast, the code in the body of an `untied` task can be executed on different threads. More specifically, when an `untied` task is resumed, it can be can be executed on any idle thread.

By default, OpenMP tasks are `tied`, unless explicitly specified as `untied`.

[1]Additional TSPs are implied at constructs `target`, `taskyield`, `taskgroup` that we do not consider in this paper for simplicity.

Task Scheduling Constraint

The OpenMP specification enforces the following constraints, namely the task scheduling constraint (**TSC**) [1]:

"*In order to start the execution of a new* `tied` *task, the new task must be a descendant of every task that is currently tied on the same thread.*"

More formal definition of **TSC** and detail illustrations are given in Sect. III-B2.

B. OpenMP System Model

1) Task System : An OpenMP task system $\mathcal{T}$ consists of n tasks $\{\tau_1, \cdots, \tau_n\}$, and each task τ_i comprise a set of sequentially ordered parts $\{P_{i0}, P_{i1}, \cdots\}$. The task system $\mathcal{T}$ can be represented as a directed acyclic graph (DAG) [2] $G = (V, E)$, where V represents the set of vertices, and E represents the set of edges. Each vertex v_{ix} in V is associated with a worst-case computation time C_{ix} and corresponds to the x-th *part* P_{ix} of task τ_i. For convenience, we also say v_{ix} belongs to τ_i. As shown in Sect. III-A3, each vertex in V is non-preemptive. The edge (v_{ix}, v_{jz}) in E denotes the precedence constraint between vertices v_{ix} and v_{jz}, indicating that v_{ix} cannot be executed unless v_{jz} has been completed. As shown in Sect. III-A2, there are three types of edges in a DAG, i.e., $E = E_1 \cup E_2 \cup E_3$:

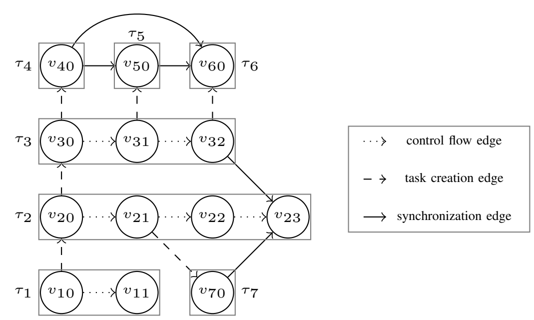


Fig. 1: DAG graph of OpenMP program in List. 1.

- **Control Flow Edges** (E_1), denoted by dotted-line arrows in Fig. 1, which model the control flow dependencies. More specifically, the control flow edges represent the sequential order of the vertices belonging to one task, i.e., for any task τ_i, there is a control flow edge between its vertices v_{ix} and $v_{i,x+1}$ $(x = 0, 1, \cdots)$.

> **Rule E1**: *The control flow edges connect vertices within the same task.*

[2]An OpenMP task system should be modelled as a DAG with *conditional branching* since the fork-join structures may be nested with the branching structures (e.g., the if-else structure). Including conditional branching semantics will the make the abstract model rather complicated (e.g., a part may be divided into several vertices in the graph). For simplicity of presentation, we assume a DAG model without conditional branching to focus on the main point of this paper, i.e., how to handle the `tied` tasks in the scheduling and analysis. The results of this paper can be extended to include conditional branching structures, the details of which is provided in technical report [15].

- **Task Creation Edges** (E_2), denoted by dashed-line arrows in Fig. 1. A parent task points its child tasks via this type of edges. In Fig. 1, (v_{31}, v_{50}) is a task creation edge.

> **Rule E2**: *The destination vertex of a task creation edge must be the* ***first*** *vertex of a task. The source and destination vertices must be in different tasks.*

- **Synchronization Edges** (E_3), denoted by solid-line arrows in Fig. 1. There are two subtypes of synchronization edges, corresponding to the `taskwait` and `depend` directives, respectively:
  - **`taskwait` Edges:** We assume that the vertex v_{ix} of τ_i corresponds to a task part following a `taskwait` directive, and that τ_j is a child task of τ_i, which is created beforehand. Then there is a `taskwait` edge from the last vertex of τ_j to v_{ix}. In Fig. 1, (v_{32}, v_{23}) and (v_{70}, v_{23}) are `taskwait` edges.

  > **Rule E3.1**: *A* `taskwait` *edge connects a child task to its parent task. The source must be the* ***last*** *vertex of the child task, and the destination can be any vertex in the parent task after creating this child.*

  - **`depend` Edges:** a `depend` edge connect two sibling tasks. For any task τ_i that has an `in` dependence on a variable and its previously created sibling τ_j that has an `out` or `inout` dependence on the same variable, there is a `depend` edge from the last vertex of τ_j to the first vertex of τ_i. Moreover, for any task τ_i that has an `out` or `inout` dependence on a variable and its previously created sibling τ_j that has an `in`, `out` or `inout` dependence on the same variable, there is a `depend` edge from the last vertex of τ_j to the first vertex of τ_i. In Fig. 1, (v_{40}, v_{50}), (v_{50}, v_{60}) and (v_{40}, v_{60}) are `depend` edges.

  > **Rule E3.2**: *The source of a* `depend` *edge must be the* ***last*** *vertex of a task, and the destination must be the* ***first*** *vertex of its sibling task.*

Particular Structure of DAG

From **Rules E1** to **E3.2**, the DAG model of an OpenMP program does not have a general topology structure. We know that generality can be a drawback from the point of view of computational efficiency, which may be typically improved by avoiding generality and exploiting particular structures. Hence in the following we investigate the particular topology structure derived by OpenMP semantics.

Lemma 1. *For any task τ, there is no edge between the non-descendant of τ and the descendant of τ.*

Proof. It is trivial to prove Lem. 1 by enumerating the three types of edges: $E = E_1 \cup E_2 \cup E_3$. By **Rule E1**, the edge in E_1 connects the vertices belonging to the same task. By **Rule E2**, an edge in E_2 connects a task τ to its child task. By **Rules E3.1** and **E3.2**, an edge in E_3 connects a task τ to its non-descendent (e.g., the parent of τ or the sibling of τ). In sum, none of the edges in E connects the non-descendant of τ to the descendant of τ. □

Lemma 2. *For any task τ and its descendant τ', if a vertex v' of τ' is the predecessor of the vertex v of τ, then the last vertex of τ' must be the predecessor of v of τ.*

Proof. It is sufficient to prove that any path from v' to v must travel the last vertex of τ'. Suppose not, there is a path λ from v' to v that does not travel the last vertex of τ'.

Let u be the first vertex of λ such that u belongs to neither τ' nor the descendent of τ'. It indicates that the immediate predecessor u' of u in λ belongs to either τ' or a descendant of τ'.

Since u belongs to a non-descendant of τ', and there is an edge between u' and u, u' cannot belong to a descendant of τ' according to Lem. 1. Therefore, u' can only belong to τ'. Thus, the original vertex u' of (u', u) belongs to τ', and the destination vertex u of (u', u) belongs to a non-descendant of τ'. We know that only the edge in E_3 can connect a task to its non-descendant. Thus, $(u', u) \in E_3$, and u belongs to either the parent of τ' or a sibling of τ'. In both of the two cases, u' must be the last vertex of τ' according to **Rules E3.1** and **E3.2**. This contradicts to the assumption. □

Lemma 3. *For any task τ and its non-descendant τ', if a vertex v' of τ' is the predecessor of the vertex v of τ, then v' of τ' must be the predecessor of the first vertex of τ.*

Proof. It is sufficient to prove that any path from v' to v must travel the first vertex of τ. Suppose not, there is a path λ from v' to v that does not travel the first vertex of τ.

Let u' be the last vertex of λ which belongs to neither τ or the descendent of τ. It implies that the immediate successor u of u' in λ belongs to either τ or a descendant of τ. We know that u' belongs to a non-descendant of τ and that there is an edge (u', u). According to Lem. 1, u cannot belongs to any descendant of τ. Therefore, u belongs to τ. More specifically, the original vertex u' of (u', u) belongs to a non-descendant of τ, and the destination vertex u of (u', u) belongs to τ. We know that only the task creation edge in E_2 and the `depend` edge can connect a non-descendant of τ to τ. Thus, (u', u) is either a task creation edge or a `depend` edge, and u' belongs to either the parent of τ or a sibling of τ. In both cases, u is the first vertex of τ according to **Rules E2** and **E3.2**. □

Additional Notations

In the following we introduce some additional notations related to the DAG task model, which will be used in analysing the response time of OpenMP task systems.

Definition 1. *A vertex v_{ix} is the* `taskwait` *vertex of τ_i if there is a* `taskwait` *edge $(v_{jz}, v_{ix}) \in E$.*

From **Rule E3.1**, the original vertex v_{jz} of the `taskwait` edge (v_{jz}, v_{ix}) must be the last vertex of task τ_j, which implies that the `taskwait` vertex cannot start its execution unless τ_j is finished. For convenience, we say τ_j *joins* to the `taskwait` vertex v_{ix}.

Definition 2. *A task τ_j is the depending task of τ_i if there is a* `taskwait` *edge from τ_j to τ_i.*

From **Rule E3.1**, the `taskwait` edge can only from a child task to the parent task. Thus, a *depending* task τ_j of τ_i implies that τ_j is a child of τ_i.

Depending Sequence. For any task sequence $\kappa = (\tau_1, \tau_2, \cdots)$, we say κ is a *depending sequence* if τ_{i+1} is the depending task of τ_i, for any adjacent tasks τ_i and τ_{i+1} in κ. Moreover, we say a depending sequence κ is *maximum* if it cannot be extended to a larger sequence, i.e., the first task of κ is not a depending task of any other task, and the last task of κ has no depending tasks.

Depth of Graph. For any maximum depending sequence κ, we count the number $\mathcal{N}_{tied}(\kappa)$ of `tied` tasks from all the tasks of κ except the last task of κ. We define the *depth* of DAG G by checking all the maximum depending sequences of G:

$$dep(G) = \max\{\mathcal{N}_{tied}(\kappa) | \kappa \text{ is maximum}\}.$$

For example, in Fig. 1, there are two depending sequences $\kappa_1 = (\tau_2, \tau_3)$ and $\kappa_2 = (\tau_2, \tau_7)$, and $\mathcal{N}_{tied}(\kappa_1) = \mathcal{N}_{tied}(\kappa_2) = 1$. The depth of G: $dep(G) = 1$. It should be emphasized that the depth $dep(G)$ only depends on the type of task τ_2. For example, if τ_2 is `untied`, then $\mathcal{N}_{tied}(\kappa_1) = \mathcal{N}_{tied}(\kappa_2) = 0$ no matter whether τ_3 and τ_7 are `tied` or not.

The calculation of $dep(\mathcal{T})$ can be found in Appendix A.

Longest Path to a `taskwait` vertex. For any `taskwait` vertex v_{ix} of τ_i, we denote by λ_{ix} the longest path such that:

- the last vertex of λ_{ix} is an immediate predecessor of v_{ix};
- λ_{ix} does not travel any vertex of τ_i.

The length of λ_{ix} is denoted as $len(\lambda_{ix})$, and the calculation of $len(\lambda_{ix})$ is given in Appendix B.

2) Schedule: Given a set of tasks $\mathcal{T} = \{\tau_1, \cdots, \tau_n\}$ and a team of threads $\mathcal{S} = \{s_1, \cdots, s_m\}$, a *schedule* can be defined as a m-dimension vector of functions $\sigma = (\sigma_1, \cdots, \sigma_m)$, and each function $\sigma_k : \mathbf{R}^+ \to \mathbf{N}$ such that $\forall t \in \mathbf{R}^+$, $\sigma_k(t) = i$, with $i > 0$, means that the thread s_k is executing task τ_{i_k} at time t, while $i = 0$ means that s_k is idle ($k = 1, \cdots, m$). Furthermore, by considering that a task τ_i comprises a set of vertices $\{v_{i0}, v_{i1}, \cdots\}$ in the DAG graph, we also define each function σ_k as follows. $\forall t \in \mathbf{R}^+$, $\sigma_k(t) = (i, x)$, with $i > 0$, means that the thread s_k is executing the vertex v_{ix} of task τ_i at time t, while $i = 0$ means that s_k is idle no matter what x equals. In the rest of the paper, we use both versions of schedule function σ_k, and without leading to confusion, we use the bold symbol $\boldsymbol{\sigma}_k$ to denote the schedule function which only returns the task index. Moreover, for convenience, σ_k (and $\boldsymbol{\sigma}_k$) also represents the sequence of vertices (and tasks) executed by the thread s_k, i.e., $v_{ix} \in \sigma_k$ means that vertex v_{ix} is executed by s_k, and $\tau_i \in \boldsymbol{\sigma}_k$ means that some vertex of τ_i is executed by s_k.

Useful Notations

Timing parameters of a vertex. By given a schedule σ, for any vertex v_{ix} of task τ_i, we define the associated timing parameters as follows.

- beginning time of v_{ix}:
$$b_{ix} = \min\{t | \sigma_k(t) = (i, x), \forall t \in \mathbf{R}^+, k \in [1, m]\}.$$
- finishing time of v_{ix}, for any $\Delta \to \mathbf{0}^+$:
$$f_{ix} = \max\{t + \Delta | \sigma_k(t) = (i, x), \forall t \in \mathbf{R}^+, k \in [1, m]\}.$$
The finishing time f_{ix} is no more than $b_{ix} + C_{ix}$.
- eligible time of v_{ix}:
$$e_{ix} = \max\{f_{jz} | \forall (v_{jz}, v_{ix}) \in E\}. \quad (1)$$
The eligible time e_{ix} of v_{ix} equals to the maximum finishing time among all the predecessors of v_{ix}. Moreover, at a time instant t, we say a vertex v_{ix} is *eligible* if $e_{ix} \le t$ and $b_{ix} > t$. Note that the beginning time b_{ix} of v_{ix} should not be less than the eligible time e_{ix}. In particular, for any vertex $v_{ix} \in V$, we say the execution of v_{ix} is *delayed* if $b_{ix} > e_{ix}$.

Current `tied` tasks for a thread. For a given time instant t and a thread s_k, we denote by $\Gamma_k(t)$ the set of tasks that are `tied` on s_k and which have not been finished at time t:

$$\Gamma_k(t) = \{\tau_i | \texttt{tied } \tau_i \in \boldsymbol{\sigma}_k \wedge \exists v_{ix} \in \tau_i, \text{ with } f_{ix} > t\}.$$

For any $\tau_i \in \Gamma_k(t)$, we say τ_i is *suspended* at time t if $\boldsymbol{\sigma}_k(t) \neq i$. In this case, there must exist a vertex v_{ix} of τ_i, with $x \ge 1$, such that $b_{ix} > t$ and $f_{i,x-1} \le t$. We say v_{ix} is the *suspending* vertex of τ_i at time t, denoted as $v_i(t) = v_{ix}$.

Response time. Finally, the response time of a task DAG $G = (V, E)$ is defined as

$$R(G) = \max\{f_{ix} | v_{ix} \in V\}.$$

We know that $\boldsymbol{\sigma}_k(t) = 0$ and $\Gamma_k(t) = \emptyset$, for any $k \in [1, m]$ and $t \ge R(G)$.

Key paths. For a given time instant t and a vertex v_{ix} of τ_i, with $b_{ix} \ge t$, the schedule σ derives the associated *key path* $\lambda_{key}(v_{ix}, t)$ as follows.

- All the vertices of $\lambda_{key}(v_{ix}, t)$ do not belong to τ_i.
- For any edge (v_{jz}, v_{ly}) of $\lambda_{key}(v_{ix}, t)$, v_{jz} is a vertex with the latest finishing time among all the predecessors of v_{ly}, and moreover, v_{jz} is completed at or after t, i.e., $f_{jz} \ge t$. In particular, the last vertex of $\lambda_{key}(v_{ix}, t)$ is the one with latest finishing time among all the predecessors of v_{ix} which is completed at or after t.

It should be emphasized that the key path $\lambda_{key}(v_{ix}, t)$ is a path whose last vertex is the immediate predecessor of v_{ix} and which does not travel any vertex of τ_i. Recall that we have used λ_{ix} to denote the longest one among such kind of paths. Thus, we have:

$$len(\lambda_{key}(v_{ix}, t)) \leq len(\lambda_{ix}),\ \forall t \in [0, R(G)] \qquad (2)$$

In some special cases, we do not predefine the target vertex v_{ix}, (i.e., $i = 0$) and moreover, let $t = 0$. The corresponding key path $\lambda_{key}(v_{0x}, 0)$ is defined as follows.

- The last vertex of $\lambda_{key}(v_{0x}, 0)$ is one with the lasted finishing time in the schedule σ.
- For any edge (v_{jz}, v_{ly}) of $\lambda_{key}(v_{0x}, 0)$, v_{jz} is a vertex with the latest finishing time among all the predecessors of v_{ly}.

We call $\lambda_{key}(v_{0x}, 0)$ as the *key path of the whole schedule* σ, and for convenience, we redefine it as λ_{key} in the rest of the paper.

Task Scheduling Constraints

A feasible schedule σ satisfies the following constraints.

> **Cons SE**: $\forall t \in \mathbf{R}^+$, and $\forall k, k' \in [1, m]$, $\boldsymbol{\sigma}_k(t) \neq \boldsymbol{\sigma}_{k'}(t)$ if $\boldsymbol{\sigma}_k(t) > 0$ or $\boldsymbol{\sigma}_{k'}(t) > 0$.

Cons SE ensures that any two different threads cannot execute the same task simultaneously. In the other words, a task should be **S**equentially **E**xecuted.

> **Cons PC**: For any $(v_{jz}, v_{ix}) \in E$, $b_{ix} \leq f_{jz}$.

Cons PC ensures the **P**recedence **C**onstraints defined by the edges in E, i.e., a vertex v_{ix} cannot start its execution unless its predecessor v_{jz} is completed.

> **Cons TSP**: For any $v_{ix} \in V$, if $v_{ix} \in \sigma_k$, then $\sigma_k(t) = (i, x)$, $\forall t \in [b_{ix}, f_{ix})$.

Cons TSP ensures that the vertex in V is non-preemptive. More specifically, the constraint enforce that once a thread s_k begins to execute vertex v_{ix} at time b_{ix}, then s_k always executes v_{ix} during interval $[b_{ix}, f_{ix})$.

> **Cons TIED**: For any vertex v_{ix} of a `tied` task $\tau_i \in \mathcal{T}$, with $x \geq 1$, $\sigma_k(t) = (i, x)$ only if $\tau_i \in \Gamma_k(t)$.

Cons TIED ensures that a `tied` task should be executed by one thread during its life cycle. More specifically, at any time t, thread s_k can execute the vertex v_{ix} of `tied` task τ_i $(x \geq 1)$ only if τ_i has been tied on s_k before time t (See the definition of $\Gamma_k(t)$).

> **Cons TSC**: For a new `tied` task τ_i, $\sigma_k(t) = (i, 0)$ if $\forall \tau_j \in \Gamma_k(t)$, τ_i is a descendant of τ_j.

Cons TSC gives a formal definition of the **T**ask **S**cheduling **C**onstraint (TSC) introduced in Sect. III-A3. This constraint enforces that a new `tied` task τ_i can be executed by thread s_k at time t only if (1) there is no unfinished task `tied` on s_k at time t, i.e., $\Gamma_k(t) = \emptyset$; or (2) τ_i is the descendant of every unfinished task that is tied on s_k. Example 1 illustrates **Cons TSC**.

In contrast, the execution of an `untied` task needs not to fulfil **Cons TIED** and **TSC**.

Example 1. *In Fig. 2, suppose that task τ_1 is* `untied` *and the other tasks $\tau_2, \cdots, \tau_5$ are* `tied`*. At time t_1, the currently* `tied` *task set of s_1 is $\Gamma_1(t_1) = \{\tau_3\}$, and the tasks τ_4 and τ_5 are both eligible at time t_1 since $e_{40} = f_{12} < t_1$ and $e_{50} = f_{11} < t_1$. Under **Cons TSC**, τ_5 can be executed by s_1 and τ_4 is delayed since τ_5 is the child of τ_3, but τ_4 is not.*

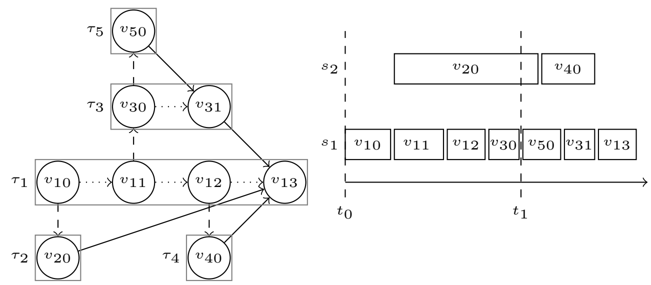


Fig. 2: An example schedule agrees with TSC.

IV. Problems with Existing Scheduling Policies

Most OpenMP implementations support two scheduling policies: Work First Scheduling (WFS) [16] and Breadth First Scheduling (BFS) [17]. Roughly speaking, WFS prefers to execute newly created tasks, while in BFS a thread tends to execute tasks that have been tied on them. When `tied` tasks are used, BFS is the only choice in practice, as WFS leads to a complete sequentialization of task executions when nested parallelism (found, for example, in programs that use recursion) is adopted. In this work, we investigate how to scheduling OpenMP programs in the presence of `tied` tasks, and thus we only focus on BFS and its extensions.

The pseudo-code of BFS is shown in Alg. 1. The algorithm is invoked at any time t when a vertex v_{ix} of τ_i completes its execution (Line 2). If τ_i has tied on a thread s_k and the immediate successor $v_{i,x+1}$ of v_{ix} is eligible (Line 3), then the scheduler assign $v_{i,x+1}$ to s_k. In this case, s_k continues the execution of τ_i (Line 4).

After that, if there is any vertex v_{jz} of τ_j that is eligible and which has not been executed at time t, the scheduler find a thread s_k for executing v_{jz}. There are two possible cases.

- τ_j is a new `tied` task (Line 7). It indicates that v_{jz} is the first vertex of τ_j. A thread s_k can execute v_{ix} only if s_k is idle (Line 9) and **Cons TSC** is fulfilled (Line 10).
- τ_j is `untied` or has been tied on a thread s_k (Line 14). In this case, s_k can execute v_{jz} if s_k is idle (Line 13).

As we know that scheduling anomalies may occur when the DAG-based task set is executed in a multiprocessor environment [13]. To void the anomalies, it is sufficient to assume that every vertex is executed a worst-case computation time.

Algorithm 1 BFS

At the current time t:
while any v_{ix} of τ_i with $f_{ix} = t$ **do**
  if $e_{i,x+1} = t$ and $\tau_i \in \Gamma_k(t)$ **then**
    assign $v_{i,x+1}$ to s_k;
  end if
  for any unexecuted v_{jz} of τ_j with $e_{jz} \leq t$ **do**
    if τ_j is a new `tied` task **then**
      assign v_{jz} to s_k **only if**
        $\sigma_k(t) = 0$; **and** for any $\tau_l \in \Gamma_k(t)$:
        τ_j is a descendant of τ_l;
    else
      assign v_{jz} to s_k **only if**
        $\sigma_k(t) = 0$; **and**
        τ_j is `untied` or $\tau_j \in \Gamma_k(t)$;
    end if
  end for
end while

Based on this assumption, the assignment of a vertex v_{ix} to a thread s_k is defined as follows.

$$\sigma_k(t) \quad := \quad (ix), \forall t \in [b_{ix}, t + C_{ix}); \tag{3}$$

$$f_{ix} \quad := \quad b_{ix} + C_{ix}; \tag{4}$$

Any schedule σ_{BFS} derived by BFS (Alg. 1) fulfils all the five constraints in Sect. III-B2, which is proved in Appendix D.

Note that **Cons SE**, **PC** and **TSP** are constraints for both of `tied` and `untied` task, while other constrains, e.g., **Cons TIED** and **TSC**, only restrict the scheduling behaviour of `tied` tasks. If all tasks are `untied`, only the constraints **Cons SE**, **PC** and **TSP** need considering. In this case, the problem degenerates to a scheduling problem on the DAG with vertex-level non-preemption (See in [13] for example). Serrano et al [4] showed that BFS is *work-conserving* when scheduling `untied` tasks, i.e., $\forall \boldsymbol{\sigma}_k \in \sigma_{BFS}$, $\boldsymbol{\sigma}_k(t) = 0$ implies that there is no eligible vertex at time t. Based on the work-conserving property, Serrano et al [4] proved that BFS algorithm derives a response time bound for an OpenMP-DAG task graph as follows.

$$R_{BFS}(G) \leq len(G) + \frac{vol(G) - len(G)}{m} \tag{5}$$

where $len(G)$ is the length of critical path in DAG G, and $vol(G)$ is the total worst-case computation time of all the vertices in G, as well as m is the number of threads.

Unfortunately, the bound in (5) is not applicable to `tied` tasks since more complicated constraints **Cons TIED** and **TSC** are considered, which makes BFS a non-work-conserving algorithm.

Nevertheless, BFS is a better choice for scheduling `tied` tasks in practice. Compared to WFS, when encountering newly created tasks, BFS does not force the thread to suspend the parent task for executing the newly encountered child task, but has a better chance to distribute `tied` tasks to different threads, which avoids a complete sequentialization of task executions. It leaves an open problem in [4] about how to bound the response time of OpenMP application with `tied` task under BFS.

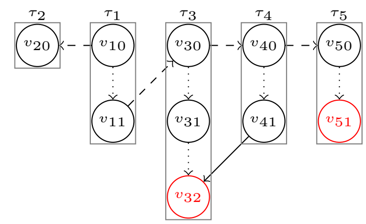


Fig. 3: An OpenMP-DAG with `tied` tasks.

In order to answer this open problem, we first give an example to show that in some cases, BFS performs as bad as WFS does. In the other words, BFS may also execute the parallel workload sequentially, and thus the general response time bound for `tied` tasks under BFS is the volume of DAG, i.e., $vol(G)$.

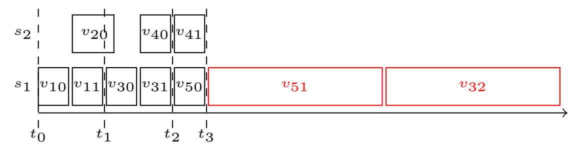


Fig. 4: The schedule σ_{BFS} for DAG in Fig. 3.

The counterexample for BFS is given in Fig. 3, where all the tasks are `tied`. The tasks τ_3 and τ_5 both have a "heavy" vertex (marked red) with WCET l, and all the other vertices are "light", whose WCETs are much smaller than l. Fig. 4 shows the schedule σ_{BFS} derived by Alg. 1. Note that thread s_1 is the only idle thread when τ_3 and τ_5 become eligible (at t_1 and t_2 respectively). As a result, τ_3 and τ_5 are tied on s_1. Consequently, the schedule σ_{BFS} has two phases. In the first phase, from t_0 to t_2, tasks are tied to s_1 and s_2. In the second phase, the "heavy" vertices execute sequentially. When the execution time of all the light vertices approaches 0, the response time of this OpenMP application approaches its total workload $vol(G) = 2l$.

From the above example, we observe that under BFS a task (e.g., τ_5) that is newly tied on a thread (e.g., s_1) will block the tasks that have been currently tied on the same thread (s_1) in the future, which, in the worst case, leads to a sequentialization of task executions. This motivates us to design a new scheduling policy to avoid this resource waste.

V. New Scheduling Policy: BFS*

Last section showed that BFS may also lead to a complete sequentialization of task executions in the presence of `tied` tasks. In this section, we develop a new scheduling policy BFS* to mitigate the resource waste problem caused by `tied` tasks. BFS* is similar to BFS in the sense of first executing currently tied tasks. But BFS* uses an enhanced

TSC constraint to prevent tying too many tasks on the same thread.

> **Cons E-TSC**: For any vertex v_{ix} belonging to a new `tied` task and a `untied` task τ_i, $\sigma_k(t) = (i, x)$ if the last vertex of τ_i is a predecessor of $v_j(t)$, $\forall \tau_j \in \Gamma_k(t)$.

Cons E-TSC ensures that the vertex newly assigned to a thread must not block the tasks that have been tied on the same thread in the future. Thus, any vertex of the task that has been tied on a thread is executed as soon as it becomes eligible. In this case, once a task has been tied to a thread, its vertices cannot be delayed. We only provide the main conclusions here, and the more formal analysis is given in Sect. V-A.

Algorithm 2 BFS*

```
1:  At the current time t:
2:  while any v_ix of τ_i with f_ix = t do
3:     if e_{i,x+1} = t and τ_i ∈ Γ_k(t) then
4:        assign v_{i,x+1} to s_k;
5:     end if
6:     for any unexecuted v_jz of τ_j with e_jz ≤ t do
7:        if τ_j is a new tied task or an untied task then
8:           assign v_jz to s_k only if
9:              σ_k(t) = 0; and for any τ_l ∈ Γ_k(t),
10:             the last vertex of τ_j is a predecessor of v_l(t);
11:       else
12:          assign v_jz to s_k only if
13:             σ_k(t) = 0; and τ_j ∈ Γ_k(t);
14:       end if
15:    end for
16: end while
```

The pseudo-code of BFS* is presented in Alg. 2, where we use **Cons E-TSC** (Line 10 in Alg. 2) instead of **Cons TSC** (Line 10 in Alg. 1). Using similar techniques in Pro. 1 to 4, we can prove that a schedule σ_{BFS^*} derived by BFS* satisfies **Cons SE**, **PC**, **TSP** and **TIED**. The following lemma shows that σ_{BFS^*} also fulfils **Cons TSC**.

Lemma 4. *σ_{BFS^*} fulfils **Cons TSC**.*

Proof. It is equal to prove that $\forall \sigma_k \in \sigma_{BFS^*}$, $\sigma_k(t) = (i, 0)$ (by assuming a `tied` task τ_i) if τ_i is the descendant of each task in $\Gamma_k(t)$. For any task $\tau_j \in \Gamma_k(t)$, we know that τ_j is suspended at time t since $\boldsymbol{\sigma}_k(t) = i$ and $i \neq j$. According to Line 10 of Alg. 2, the last vertex of τ_i is the predecessor of $v_j(t)$, the suspending vertex of τ_j at time t. We consider the following two cases:

Case 1. τ_i is a descendant of τ_j. In this case, **Cons TSC** is fulfilled.

Case 2. τ_i is a non-descendant task of τ_j. As we assumed that the last vertex of τ_i is the predecessor of $v_j(t)$ of τ_j, according to Lem. 3, the last vertex of τ_i must be the predecessor of the first vertex of τ_j. It indicates that τ_j cannot start the execution unless τ_i has been completed. This contradicts to the assumption that τ_j is suspended and τ_i is a new `tied` task. □

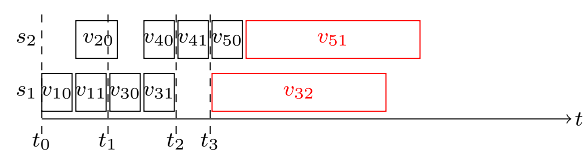


Fig. 5: The schedule σ_{BFS^*} for DAG in Fig. 3.

Fig. 5 shows the resulting schedule σ_{BFS^*} of the example in Fig. 3. At time t_1, τ_3 is suspended and s_1 becomes idle. However, the newly created task τ_5 cannot be assigned to s_1 since τ_5 is not the predecessor of τ_3 (so **ESC-(c)** is not satisfied). After a short idle period $[t_1, t_2]$, the depending task τ_4 of τ_3 is finished and v_{32} becomes eligible. s_1 continues to execute τ_3, and meanwhile, s_2 starts the execution of τ_5. In this example, the "heavy" vertices v_{32} and v_{51} are distributed to different threads.

*A. Properties of BFS**

In the following we introduce some properties of BFS*, which will be useful to derive the response time bounds in the next section.

The first one is about *delayed* vertices. Recall that a vertex v_{ix} is *delayed* if $b_{ix} > e_{ix}$. The following lemma implies that once a `tied` task starts the execution, none of its vertices can be delayed during the following scheduling process.

Lemma 5. *For any `tied` task τ_i, vertex v_{ix} is the first vertex of τ_i if $b_{ix} > e_{ix}$ in a schedule σ_{BFS^*}.*

Proof. This is proved by contradiction. Suppose that the vertex v_{ix} which is not the first vertex of a `tied` task τ_i, has a beginning time $b_{ix} > e_{ix}$ in a schedule σ_{BFS^*}. There must be a time instant $t \in (e_{ix}, b_{ix})$, and the `tied` task τ_i is suspended at time t. Let $\tau_i \in \boldsymbol{\sigma}_k$, we have $\tau_i \in \Gamma_k(t)$, with $v_i(t) = v_{ix}$. We consider the following two cases.

If $\boldsymbol{\sigma}_k(t) = 0$. v_{ix} is not eligible at t since we have assumed that v_{ix} is not executed by the idle thread s_k that has tied τ_i at t. More specifically, $e_{ix} > t$.

Otherwise. suppose that $\sigma_k(t) = (j, z)$, We know that $b_{jz} \leq t \leq b_{ix}$ and $b_{jz} \geq f_{i,x-1}$. Thus, $\tau_i \in \Gamma_k(b_{jz})$. $\sigma_k(b_{jz}) = (j, z)$ implies that the last vertex of τ_j is the predecessor of v_{ix} as **Cons E-TSC** is fulfilled. It indicates that v_{ix} is not eligible at t, since τ_j has not been completed at t. Thus, $e_{ix} > t$.

The above cases both contradict with the assumption that $e_{ix} < t$, which completes the proof. □

The second property of BFS* is about suspended tasks. Recall that a `tied` task $\tau_i \in \Gamma_k(t)$ is suspended at time t if $\boldsymbol{\sigma}_k(t) \neq i$, and τ_i will resume at its suspending vertex $v_i(t)$. Moreover, recall that the `taskwait` vertex of a task if it is pointed by `taskwait` edges. The following lemma shows that a `tied` task can only be suspended at its `taskwait` vertices.

Lemma 6. *In a schedule σ_{BFS^*}, with $\tau_i \in \Gamma_k(t)$ and $\sigma_k(t) \neq i$, $\upsilon_i(t) = v_{ix}$ only if v_{ix} is a* `taskwait` *vertex.*

Proof. Suppose that v_{ix} of τ_i is not a `taskwait` vertex, i.e., only edge $(v_{i,x-1}, v_{ix})$ points to v_{ix}. At time t, the `tied` task τ_i is suspended, with $\upsilon_i(t) = v_{ix}$. Let $\sigma_k(t) = (j, z)$, with $i \neq j$. We know that the execution of v_{jz} begins after the completion of $v_{i,x-1}$ and completes before the start of v_{ix}, i.e., $f_{i,x-1} \leq b_{jz} \leq b_{ix}$. Thus, $\tau_i \in \Gamma_k(b_{jz})$.

On the one hand, $\sigma_k(b_{jz}) = (j, z)$ implies that the last vertex of τ_j is the predecessor of v_{ix} as **Cons E-TSC** is fulfilled.

On the other hand, since $b_{jz} \geq f_{i,x-1}$, the last vertex of τ_j is not the predecessor of $v_{i,x-1}$.

In sum, v_{ix} is reachable from the last vertex of τ_j via a path λ that does not travel any vertex in τ_i. Let (v_{ly}, v_{ix}) be the last edge in λ, we have $v_{ly} \neq v_{i,x-1}$, which contradicts with the assumption. □

VI. Response Time Bounds

For any DAG G, we use BFS* algorithm to schedule it, and let $R(G)$ be the response time of G derived by a schedule σ_{BFS^*}. Without loss of generality, we assume that the schedule σ_{BFS^*} begins at time 0. For any time instant $t \in [0, R(G)]$ and any thread $s_k \in \mathcal{S}$, the value of schedule function $\boldsymbol{\sigma}_k(t)$ equals either 0 or not 0. We construct the corresponding index functions as follows.

For any time $t \in [0, R(G)]$ and $k \in [1, m]$:

$$f_{busy}^k(t) = \begin{cases} 1 & \boldsymbol{\sigma}_k(t) \neq 0 \\ 0 & else \end{cases};$$

Moreover, we let $f_{idle}^k(t) = 1 - f_{busy}^k(t)$. The cumulative values of these functions are defined as follows.

$$F_{busy}^k = \int_0^{R(G)} f_{busy}^k(t)\mathrm{d}t; \; F_{idle}^k = \int_0^{R(G)} f_{idle}^k(t)\mathrm{d}t$$

where F_{busy}^k and F_{idle}^k respectively define the busy and idle time of thread s_k. Moreover, $F_{busy} = \sum_{k=1}^m F_{busy}^k$ and $F_{idle} = \sum_{k=1}^m F_{idle}^k$ respectively denote the total busy and idle time of all the threads.

The response time $R(G)$ can be represented as follows.

$$R(G) = \frac{F_{busy} + F_{idle}}{m}. \tag{6}$$

We know that

$$F_{busy} \leq vol(G) \tag{7}$$

In the following we focus on the upper bound of F_{idle}.

By given a key path λ_{key} of the schedule σ_{BFS^*}, we define the index function below:

$$g(t) = \begin{cases} 1 & \text{a vertex of } \lambda_{key} \text{ is executing at } t \\ 0 & else \end{cases}$$

Then the total idle time F_{idle} can be rewritten as follows.

$$\begin{aligned} F_{idle} &= \sum_{k=1}^m \int_0^{R(G)} f_{idle}^k(t)\mathrm{d}t \\ &= \sum_{k=1}^m \int_0^{R(G)} [g(t) + (1 - g(t))] f_{idle}^k(t)\mathrm{d}t \end{aligned}$$

Let $h(t) = (1 - g(t))$, we rewrite F_{idle} as summation of two parts:

$$F_{idle} = F_{idle}^{key} + F_{idle}^{nokey} \tag{8}$$

where

$$F_{idle}^{key} = \sum_{k=1}^m \int_0^{R(G)} g(t) f_{idle}^k(t)\mathrm{d}t \tag{9}$$

$$F_{idle}^{nokey} = \sum_{k=1}^m \int_0^{R(G)} h(t) f_{idle}^k(t)\mathrm{d}t \tag{10}$$

Since $g(t)$ and $f_{idle}^k(t)$ are both non-negative for any $t \in [0, R(G)]$, and from (9), the first item of RHS in (8) has an upper bound below.

$$F_{idle}^{key} \leq \int_0^{R(G)} g(t) \sum_{k=1}^m f_{idle}^k(t)\mathrm{d}t \tag{11}$$

Note that $g(t)$ equals either 1 or 0, and when $g(t) = 1$, $\sum_{k=1}^m f_{idle}^k(t) \leq m - 1$ as at least one thread at time t is busy for executing the vertex in the key path λ_{key}. Therefore, (11) can further derives the following inequality.

$$F_{idle}^{key} \leq (m - 1) \int_0^{R(G)} g(t)\mathrm{d}t \tag{12}$$

We know that $\int_0^{R(G)} g(t)\mathrm{d}t$ equals the length $len(\lambda_{key})$ of the key path λ_{key}. Thus, from (12), we have:

$$F_{idle}^{key} \leq (m - 1) len(\lambda_{key}) \tag{13}$$

In the following we derive the upper bound for F_{idle}^{nokey}, the second item of RHS in (8), by two different methods, using which we can finally obtain two response time bounds. Before enter into the details, we first introduce a frequently used expression below.

> "**at time** t **when** $h(t) = 1$" $\models$ "at time t when none of the vertices in key path λ_{key} is being executed".

According to the definition of the index function $h(t)$, the LHS and RHS of the above expression equals each other.

A. First Upper Bound for F_{idle}^{nokey}

Since $h(t)$ and $f_{idle}^k(t)$ are both non-negative for any $t \in [0, R(G)]$, and from (10), an upper bound of F_{idle}^{nokey} is as follows.

$$F_{idle}^{nokey} \leq \int_0^{R(G)} h(t) \sum_{k=1}^m f_{idle}^k(t)\mathrm{d}t \tag{14}$$

We know that $h(t)$ is either 1 or 0 at any time t, and we denote by m_{idle}^{nokey} the maximum value of $\sum_{k=1}^{m} f_{idle}^{k}(t)$ for all the time t when $h(t) = 1$. Then (14) can further derive the following inequality.

$$F_{idle}^{nokey} \leq m_{idle}^{nokey} \int_{0}^{R(G)} h(t)\mathrm{d}t \tag{15}$$

Moreover, since $\int_{0}^{R(G)} h(t)\mathrm{d}t = R(G) - len(\lambda_{key})$, and from (15), we have:

$$F_{idle}^{nokey} \leq m_{idle}^{nokey}(R(G) - len(\lambda_{key})) \tag{16}$$

Note that m_{idle}^{nokey} denotes the maximum number of idle threads at all the time when no vertex of key path λ_{key} is being executed. The following theorem gives an upper bound of m_{idle}^{nokey}.

Theorem 1. *For any task DAG G scheduled by BFS*,*

$$m_{idle}^{nokey} \leq \frac{dep}{1+dep} m \tag{17}$$

where $dep = \min\{dep(G), m-1\}$.

To prove this theorem, it is sufficient to show that at a time t when $h(t) = 1$, the idle threads can be divided into at least $\frac{m}{1+dep}$ disjoint subsets, such that: each subset of idle threads corresponds to a dedicated busy thread. In the following we show such a division of idle threads as follows. We first show the property of an idle thread (See in Lem. 7).

Lemma 7. *At time t when $h(t) = 1$, $\Gamma_k(t) \neq \emptyset$ for any idle thread s_k.*

The proof of Lem. 7 is given in Appendix E.

Lem. 7 shows that at time t when $h(t) = 1$ any idle thread s_k has a non-empty $\Gamma_k(t)$, and all the tasks in $\Gamma_k(t)$ are suspended. For any idle thread s_k, we denote by $\tau(s_k)$ the last `tied` task suspended on s_k (before t):

$$\tau(s_k) = \arg\max\{t' | \boldsymbol{\sigma}_k(t') \in \Gamma_k(t), t' < t\}.$$

In order to obtain a division of all the idle threads, we define a relation $\mathcal{H}$ as follows.

Definition 3. *For any two idle threads s_k and s_l, $s_k\mathcal{H}s_l$ if there is a depending sequence from $\tau(s_k)$ to $\tau(s_l)$.*

$\mathcal{H}$ divides the idle threads into several disjoint subsets $\mathcal{H} = \{\mathcal{H}_1, \mathcal{H}_2, \cdots\}$[3] such that $\forall s_k, s_l \in \mathcal{H}_i$, $s_k\mathcal{H}s_l$ or $s_l\mathcal{H}s_k$. Moreover, $\mathcal{H}_i$ can also be represented as a sequence of threads $\mathcal{H}_i = (s_1, s_2, \cdots)$ such that for any $s_l, s_k \in \mathcal{H}_i$, $l < k$ if $s_l\mathcal{H}s_k$. We denote by $s(\mathcal{H}_i)$ the last thread in the sequence $\mathcal{H}_i$. Furthermore, we denote $\mathcal{T}_i$ the task set that contains the last suspended tasks of all the threads in $\mathcal{H}_i$. We denote by $\kappa(\mathcal{H}_i)$ the depending sequence that ends at $\tau(s(\mathcal{H}_i))$ and which contains all the tasks of $\mathcal{T}_i$. We say $\kappa(\mathcal{H}_i)$ is the depending sequence corresponding to $\mathcal{H}_i$.

The following lemma shows that each subset $\mathcal{H}_i$ corresponds to at least one busy thread.

[3]For convenience, the division of idle threads is also denoted by $\mathcal{H}$.

Lemma 8. *At time t when $h(t) = 1$, for each $\mathcal{H}_i \in \mathcal{H}$ and any $s_k \in \mathcal{H}_i$, the last suspended task $\tau(s_k)$ of s_k must has a created descendant τ_l such that:*

- *there is a depending sequence from $\tau(s_k)$ to τ_l*[4].
- *τ_l is being executed by a busy thread at time t.*

Proof. Suppose not, every created descendant of $\tau(s_k)$ is either unexecuted or suspended at time t. Among all the created descendant of $\tau(s_k)$, we focus on the one, denoted as τ_j such that:

- there is a depending sequence from $\tau(s_k)$ to τ_j.
- τ_j is neither pointed by its created siblings nor has any `taskwait` vertex.

There are two possible cases.

Case 1. τ_j is unexecuted at time t. Since τ_j has been created and no created sibling of τ_l points to τ_l, we know that τ_l is eligible at time t. In the following we show that the assignment of τ_j to s_k does not violate **Cons E-TSC**.

We have assumed that there is a depending sequence κ from $\tau(s_k)$ to τ_j, and without loss of generality, we assume that $\tau(s_k)$ and τ_j are adjacent in κ, i.e., τ_j is the depending task of $\tau(s_k)$. In this case, $\tau(s_k)$ should be suspended unless τ_j is completed. It implies that the last vertex of τ_j is the predecessor of the suspending vertex of $\tau(s_k)$. Moreover, since $\tau(s_k)$ is the last suspended task of s_k at or before t, according to Line 10 of Alg. 2, the last vertex of $\tau(s_k)$ is the predecessor of the suspending vertex of any task τ_l in $\Gamma_k(t')$, where t' is the beginning time of the vertex of $\tau(s_k)$ that is last completed before t. It further indicates that the last vertex of τ_j is the predecessor of the suspending vertex of any task in $\Gamma_k(t')$. Finally, since $\Gamma_k(t) = \Gamma_k(t') \cup \{\tau(s_k)\}$, we know that the last vertex of τ_j is the predecessor of the suspending vertex of any task in $\Gamma_k(t)$, which coincides with **Cons E-TSC**.

Therefore, the assignment of τ_j to s_k does not violate **Cons E-TSC**. According to Line 10 of Alg. 2 τ_j can be executed at time t, which leads to a contradiction.

Case 2. τ_j is suspended at time t. Since τ_j has no `taskwait` vertex, τ_j cannot be suspended according to Lem. 6. This leads to a contradiction.

In sum, at least one created descendant of $\tau(s_k)$ is being executed at time t, which completes the proof. □

For any $\mathcal{H}_i \in \mathcal{H}$, we focus on the last thread $s(\mathcal{H}_i)$ of $\mathcal{H}_i$, and let $s(\mathcal{H}_i) = s_k$. According to Lem. 8, at time t when $h(t) = 1$, the last suspended task $\tau(s_k)$ of s_k has a created descendant τ_l which is executed by a busy thread s_l. For convenience, we say s_l is the *busy thread corresponded* to $\mathcal{H}_i$.

Lemma 9. *At time t when $h(t) = 1$, each $\mathcal{H}_i \in \mathcal{H}$ corresponds to a dedicated busy thread.*

Proof. Suppose not, $\mathcal{H}$ contains two subsets $\mathcal{H}_i$ and $\mathcal{H}_j$ which shares the same corresponding busy thread. More specifically,

[4]Such τ_l must exist. Otherwise, $\tau(s_k)$ has no depending tasks, implying that there is no `taskwait` vertex in $\tau(s_k)$, and thus $\tau(s_k)$ cannot be suspended according to Lem. 6. This contradicts with the fact that $\tau(s_k)$ is suspended on s_k.

by τ_q we denote the task that is executed by this busy thread. Moreover, we respectively denote by s_k and s_l the last threads of $\mathcal{H}_i$ and $\mathcal{H}_j$, i.e., $s_k = s(\mathcal{H}_i)$ and $s_l = s(\mathcal{H}_j)$. The last suspended tasks of s_k and s_l are $\tau(s_k)$ and $\tau(s_l)$ respectively. According to Lem. 8, we know that:

- τ_q is the descendant of $\tau(s_k)$ and $\tau(s_l)$;
- there is a depending sequence κ_k from $\tau(s_k)$ to τ_q;
- there is a depending sequence κ_l from $\tau(s_l)$ to τ_q.

Moreover, we denote by $\kappa(\mathcal{H}_i)$ and $\kappa(\mathcal{H}_j)$ the depending sequences corresponding to $\mathcal{H}_i$ and $\mathcal{H}_j$ respectively. Since $\mathcal{H}_i$ and $\mathcal{H}_j$ are disjoint, we have $\kappa(\mathcal{H}_i) \oplus \kappa(\mathcal{H}_j) \neq \emptyset$ (some task contained in $\kappa(\mathcal{H}_i)$ does not belong to $\kappa(\mathcal{H}_j)$, and vice versa).

Since $\tau(s_k)$ is the last task of $\kappa(\mathcal{H}_i)$ and is the first task of κ_k, we can connect these two sequences and obtain a larger one $\kappa_1 = \kappa(\mathcal{H}_i) + \kappa_k$, where "$+$" is the symbol representing the connection of two sequences. Likewise, we denote $\kappa_2 = \kappa(\mathcal{H}_j) + \kappa_l$. Since $\kappa(\mathcal{H}_i) \oplus \kappa(\mathcal{H}_j) \neq \emptyset$, we have $\kappa_1 \oplus \kappa_2 \neq \emptyset$. On the other hand, κ_1 and κ_2 end at the same task τ_q. We let τ_p be the nearest task τ_q that is shared by both κ_1 and κ_2. By τ_x we denote the immediate predecessor of τ_p in the sequence κ_1, and let τ_y be the immediate predecessor of τ_p in the sequence κ_2. We know that τ_p is the depending task of two different tasks τ_x and τ_y. According to the definition of depending tasks, we know that τ_p is the first level child of both τ_x and τ_y, which leads to a contradiction. □

Based on Lem. 8 and 9, we complete the proof of Thm. 1 as follows.

proof of Thm. 1. Suppose that at a time when the key path λ_{key} is not executed, the number of the idle threads is m_{idle}^{nokey}. We give a division $\mathcal{H}$ for these m_{idle}^{nokey} idle threads, i.e., $\sum_{\mathcal{H}_i \in \mathcal{H}} |\mathcal{H}_i| = m_{idle}^{nokey}$.

For any $\mathcal{H}_i \in \mathcal{H}$, in the following we show that $|\mathcal{H}_i| \leq dep(G)$. Let $\kappa(\mathcal{H}_i)$ be the depending sequence corresponding to $\mathcal{H}_i$, and we denote by κ_i the maximum depending sequence that fully contains $\kappa(\mathcal{H}_i)$. Recall that the last task of a maximum depending sequence has no depending task. It implies that the last task of κ_i is not the last suspended task of any thread in $\mathcal{H}_i$. Thus, the number of threads in $\mathcal{H}_i$ is no more than the number of the tied tasks in κ_i (except the last task of κ_i), i.e., $|\mathcal{H}_i| \leq \mathcal{N}_{tied}(\kappa_i)$. Moreover, since $\mathcal{N}_{tied}(\kappa_i) \leq dep(G)$, we have $|\mathcal{H}_i| \leq dep(G)$.

Since any $\mathcal{H}_i$ of $\mathcal{H}$ has at most $\min\{dep(G), m_{idle}^{nokey}\}$ threads, and $\sum_{\mathcal{H}_i \in \mathcal{H}} |\mathcal{H}_i| = m_{idle}^{nokey}$, we have:

$$|\mathcal{H}| \geq \frac{m_{idle}^{nokey}}{\min\{dep(G), m_{idle}^{nokey}\}} \tag{18}$$

According to Lem. 8 and 9, each subset of $\mathcal{H}$ corresponds to a dedicated busy thread. Thus, there are at least $|\mathcal{H}|$ busy threads. We have:

$$m_{idle}^{nokey} + |\mathcal{H}| \leq m \tag{19}$$

Combine (18) and (19), we have

$$m_{idle}^{nokey} \leq \frac{\min\{dep(G), m_{idle}^{nokey}\}}{1 + \min\{dep(G), m_{idle}^{nokey}\}} m \tag{20}$$

Moreover, since $m_{idle}^{nokey} \leq m - 1$ (there is at least one busy thread at any time $t \in [0, R(G)]$), (20) implies (17). □

According to Thm. 1, and from (16), we derive the first upper bound for F_{idle}^{nokey} below.

$$F_{idle}^{nokey} \leq (R(G) - len(\lambda_{key})) \frac{dep}{1 + dep} m \tag{21}$$

B. Second Upper Bound for F_{idle}^{nokey}

By W_{tied} we denote the set of taskwait vertices that belong to tied tasks. For each taskwait vertex $v_{ix} \in W_{tied}$, and each thread s_k, we denote an index function $w_{ix}^k(t)$, such that at time t, $w_{ix}^k(t) = 1$ if:

- s_k is idle, i.e., $f_{idle}^k(t) = 1$; and
- τ_i is the last task of $\Gamma_k(t)$, i.e., $\tau_i = \tau(s_k)$; and
- the suspending vertex of τ_i is $\upsilon_i(t) = v_{ix}$.

Otherwise, $w_{ix}^k(t) = 0$.

For any time t when $h(t) = 1$, according to Lem. 7, there must be a tied task suspended on s_k if s_k is idle (or equally, $f_{idle}^k(t) = 1$). Moreover, a tied task is suspended at its taskwait vertex (Lem. 6). Thus, we have:

$$f_{idle}^k(t) = \sum_{v_{ix} \in W_{tied}} w_{ix}^k(t), \ \forall k \in [1, m] \tag{22}$$

Combine (22) to (10), we have:

$$F_{idle}^{nokey} = \sum_{k=1}^{m} \int_0^{R(G)} h(t) \sum_{v_{ix} \in W_{tied}} w_{ix}^k(t) \mathrm{d}t \tag{23}$$

Since $h(t)$ and $w_{ix}^k(t)$ are non-negative, we have:

$$F_{idle}^{nokey} \leq \int_0^{R(G)} h(t) \sum_{v_{ix} \in W_{tied}} w_{ix}(t) \mathrm{d}t \tag{24}$$

where $w_{ix}(t) = \sum_{k=1}^{m} w_{ix}^k(t)$. We know that $w_{ix}(t) = 1$ if τ_i is the last tied task suspended on some thread at t, and the suspending vertex of τ_i is $\upsilon_i(t) = v_{ix}$. Otherwise, $w_{ix}(t) = 0$.

Lemma 10. *For any time t when $h(t) = 1$, $w_{ix}^k(t) = 0$ if $v_{ix} \in \lambda_{key}$.*

Proof. Suppose not. At time t when $h(t) = 1$, we let $w_{ix}^k(t) = 1$ for a taskwait vertex v_{ix} that belongs to W_{tied} and which is contained in the key path λ_{key}. More specifically, at time t:

- a vertex v_{jz} of the key path λ_{key} is delayed;
- s_k is idle, and τ_i is the last tied task suspended on s_k; and the suspending vertex of τ_i is $\upsilon_i(t) = v_{ix}$;
- v_{ix} of τ_i is contained in λ_{key}.

Since v_{jz} and v_{ix} are both in λ_{key}, there are two cases.

Case 1. v_{ix} is the predecessor of v_{jz}. On the one hand, since v_{ix} is unexecuted at time t, it implies that its successor v_{jz} is not eligible at time t, i.e., $e_{jz} > t$. On the other hand, v_{jz} is delayed at time t, we know that $b_{jz} > t > e_{ix}$. This leads to a contradiction.

Case 2. v_{jz} of τ_j is the predecessor of v_{ix} of τ_i.

- If τ_j is a descendant of τ_i, the last vertex of τ_j is the predecessor of v_{ix} according to Lem. 2. Moreover, according to Lem. 5, τ_j is a new `tied` task, or an `untied` task. The assignment of v_{jz} to s_k does not violate `Cons E-TSC`, and based on Lines 7 to 10 of Alg. 2, v_{jz} can be executed by s_k at time t, which contradicts with the assumption.

- Otherwise, τ_j is a non-descendant of τ_i. According to Lem. 3, v_{jz} is the predecessor of the first vertex of τ_i. It implies that τ_i cannot start its execution unless v_{jz} is completed, i.e., $b_{i0} > f_{jz} > t$. More specifically, τ_i has not began its execution at time t, which contradicts with the assumption that τ_i has been suspended at time t.

In sum, the above cases both lead to contradictions. □

According to Lem. 10, we have: for any $t \in [0, R(G)]$,

$$h(t) \sum_{v_{ix} \in W_{tied}} w_{ix}(t) \leq \sum_{v_{ix} \in W_{tied}^{nokey}} w_{ix}(t) \tag{25}$$

where W_{tied}^{nokey} represents the set of `taskwait` vertices that belong to `tied` tasks and which is not contained in the key path λ_{key}.

Combine (25) and (24), we have:

$$F_{idle}^{nokey} \leq \int_0^{R(G)} \sum_{v_{ix} \in W_{tied}^{nokey}} w_{ix}(t) \mathrm{d}t \tag{26}$$

$$= \sum_{v_{ix} \in W_{tied}^{nokey}} W_{ix} \tag{27}$$

where $W_{ix} = \int_0^{R(G)} w_{ix}(t)\mathrm{d}t$, which equals the total amount of the idle time associated with a thread when the last `tied` task suspended on the idle thread is τ_i and when the suspending vertex of τ_i is v_{ix}. Moreover, since for any $v_{ix} \in W_{tied}$, $w_{ix}(t) = 1$ implies that $t \in [f_{i,x-1}, b_{ix}]$, we have:

$$W_{ix} = \int_{f_{i,x-1}}^{b_{ix}} w_{ix}(t)\mathrm{d}t \tag{28}$$

For any continuous interval $[t_1, t_2] \in [f_{i,x-1}, b_{ix}]$, we say $[t_1, t_2]$ belongs to W_{ix} if $w_{ix}(t) = 1$, $\forall t \in [t_1, t_2]$. Fig. 6 shows the idle intervals belonging to W_{ix}. Note that $W_{i,x}$ may consist of a continuous time interval (e.g., $[t_1, t_2]$ in Fig. 6(a)), or several disjoint time intervals (e.g., $[t_1, t_2]$ and $[t_3, t_4]$ in Fig. 6(b)).

(a) a continues idle interval (b) disjoint idle intervals

Fig. 6: Idle intervals belonging to W_{ix}.

The following lemma gives an upper bound for W_{ix}. Recall that $\lambda_{key}(v_i x, t)$ denotes the key path that ends at $v_i x$ and begins after t.

Lemma 11. $W_{ix} \leq len(\lambda_{key}(v_{ix}, f_{i,x-1}))$, $\forall v_{ix} \in W_{tied}^{nokey}$.

Proof. Eq (28) indicates that the life time of the key path $\lambda_{key}(v_{ix}, f_{i,x-1})$ which ranges from the beginning time $f_{i,x-1}$ to the ending time b_{ix} fully covers all the idle time periods belonging to W_{ix}. Therefore, it is sufficient to prove $W_{ix} \leq len(\lambda_{key}(v_{ix}, f_{i,x-1}))$ by showing that: *at any time t when some vertex of $\lambda_{key}(v_{ix}, f_{i,x-1})$ is delayed, $w_{ix}(t) = 0$ always holds.*

Suppose not. At time t:

- a vertex v_{jz} of τ_j that belongs to key path $\lambda_{key}(v_{ix}, f_{i,x-1})$ is delayed;
- $w_{ix}(t) = 1$, implying that τ_i has been tied on a idle thread s_k, and moreover, τ_i is the last suspended task of $\Gamma_k(t)$, with suspending vertex $v_i(t) = v_{ix}$.

From the definition of the key path $\lambda_{key}(v_{ix}, f_{i,x-1})$, v_{jz} is the predecessor of v_{ix}. In the following we show that τ_j is the descendant of τ_i. Otherwise, we suppose that τ_j is the non-descendant of τ_i. According to Lem. 3, v_{jz} is the predecessor of the first vertex of τ_i. Moreover, from the proof of Lem. 3, we know that any path from v_{jz} to v_{ix} must travel the first vertex of τ_i, which contradicts with the definition of $\lambda_{key}(v_{ix}, f_{i,x-1})$.

Therefore, τ_j is the descendant of τ_i. According to Lem. 2, we have:

(s1) The last vertex of τ_j is the predecessor of v_{ix}.

Moreover, since τ_i is the last suspended task of $\Gamma_k(t)$, and the latest finished vertex of τ_i at or before t is $v_{i,x-1}$, according to Line 10 of Alg. 2 the assignment of $v_{i,x-1}$ to s_k implies that:

(s2) The last vertex of τ_i is the predecessor of $v_l(b_{i,x-1})$, $\forall \tau_l \in \Gamma_k(b_{i,x-1})$.

From (s1) and (s2), we have:

(s3) The last vertex of τ_j is the predecessor of $v_l(b_{i,x-1})$, $\forall \tau_l \in \Gamma_k(b_{i,x-1})$.

During the interval $[b_{i,x-1}, t]$, s_k does not execute any `tied` task except τ_i. Thus, $\Gamma_k(t) = \Gamma_k(b_{i,x-1}) \cup \{\tau_i\}$ and $v_l(t) = v_l(b_{i,x-1})$ for any $\tau_l \in \Gamma_k(b_{i,x-1})$. From (s1) and (s3), we have:

($*$) For any $\tau_i \in \Gamma_k(t)$, the last vertex of τ_j is the predecessor of $v_i(t)$.

According to Lem. 5, τ_j is a new `tied` task or an `untied` task. ($*$) implies that the assignment of v_{jz} to s_k does not violate **Cons E-TSC**. According to Line 10 of Alg. 2, v_{jz} can be assigned to s_k at time t, which contradicts with the assumption that v_{jz} is delayed at time t. □

According to Lem. 11, and from (2) and (27), we have:

$$F_{idle}^{nokey} \leq \sum_{v_{ix} \in W_{tied}^{nokey}} len(\lambda_{key}(v_{ix}, f_{i,x-1})) \tag{29}$$

$$\leq \sum_{v_{ix} \in W_{tied}^{nokey}} len(\lambda_{ix}) \tag{30}$$

where λ_{ix} is the longest path that points to v_{ix} and which does not travel any vertex of τ_i.

Moreover, we let W_{tied}^{key} be the set of `taskwait` vertices that belong to `tied` task and which are contained in the key path λ_{key}. Then we have: $W_{tied}^{nokey} = W_{tied} - W_{tied}^{key}$. Eq (30) can be rewritten as follows.

$$F_{idle}^{nokey} \leq \sum_{v_{ix} \in W_{tied}} len(\lambda_{ix}) - \sum_{v_{ix} \in W_{tied}^{key}} len(\lambda_{ix}) \quad (31)$$

Response Time Bounds

Theorem 2. *$R^{ub}(G)$ is a response time bound of a OpenMP-DAG G scheduled by BFS* on m threads:*

$$R^{ub}(G) = len(G) + \frac{1 + dep}{m}(vol(G) - len(G)) \quad (32)$$

where $dep = \min\{dep(G), m - 1\}$.

Proof. Combine (7), (13) and (21) into (6) and (8), we have:

$$R(G) \leq \frac{1 + dep}{m} vol(G) + (1 - \frac{1 + dep}{m}) len(\lambda_{key}) \quad (33)$$

Since $\frac{1+dep}{m} \leq 1$ and $len(\lambda_{key}) \leq len(G)$, inequality (33) implies inequality (32). □

The bound in (32) is simple, but may grossly overestimate the response time. In the following we derive a more precise response time bound. Before enter into details, we first define a virtual computation time for each vertex v_{ix} in G:

$$C_{ix}^{v} = \begin{cases} (m-1)C_{ix} - len(\lambda_{ix}) & v_{ix} \in W_{tied} \\ (m-1)C_{ix} & else \end{cases} \quad (34)$$

By Λ we denote the set of paths in G that start with a vertex without ingoing edges and which ends at a vertex without outgoing edges. The virtual length of G is defined as follows.

$$len^{v}(G) = \max\{\sum_{v_{ix} \in \lambda} C_{ix}^{v} | \lambda \in \Lambda\} \quad (35)$$

The calculation of $len^{v}(G)$ is given in Appendix C.

Theorem 3. *$R^{ub}(G)$ is a response time bound of a OpenMP-DAG G scheduled by BFS* on m threads:*

$$R^{ub}(G) = \frac{vol(G) + len^{v}(G) + \sum_{v_{ix} \in W_{tied}} len(\lambda_{ix})}{m} \quad (36)$$

Proof. Combine (13) and (31) into (8), we have:

$$\begin{aligned} F_{idle} \leq \; & (m-1) len(\lambda_{key}) - \sum_{v_{ix} \in W_{tied}^{key}} len(\lambda_{ix}) \\ & + \sum_{v_{ix} \in W_{tied}} len(\lambda_{ix}) \end{aligned}$$

According to the definition of virtual computation times, the first and second items of RHS in the above inequality can be rewritten as follows.

$$len^{v}(\lambda_{key}) = \sum_{v_{ix} \in \lambda_{key}} C_{ix}^{v}$$

As the key path λ_{key} starts with a vertex without ingoing edges and ends at a vertex without outgoing edges, we have $len^{v}(\lambda_{key}) \leq len^{v}(G)$. Then,

$$F_{idle} \leq len^{v}(G) + \sum_{v_{ix} \in W_{tied}} len(\lambda_{ix}) \quad (37)$$

Combine (37) and (7) into (6), we obtain (36). □

Complexity of computing bounds. We know that the volume $vol(G)$ and $len(G)$ can be respectively calculated within $O(|V|)$ and $O(|E|)$ times. According to Appendix A, the calculation of $dep(G)$ terminates within $O(n)$ times, where n is the number of tasks. Thus, the bound in (32) can be calculated within $O(n + |V| + |E|)$ times. Moreover, the calculation of $len(\lambda_{ix})$ terminates within $O(|V|)$ times, and the calculation of $len^{v}(G)$ terminates within $O(|E|)$ times (See in Appendixes B and D respectively). Therefore, the calculation of bound in (36) terminates within $O((|W_{tied}|+1)|V|+2|E|)$ times. In general, the task number n is usually much smaller than the number of edges, i.e., $n \leq |E|$. Thus, intuitively, the bound in (32) is more complicated to be computed. In the next section, we will evaluate the tightness of these two bounds, and show that the bound in (36) is much tighter, and is more applicable for realistic benchmarks.

VII. Evaluation

In this section, we evaluate the tightness of the two response time bounds with both randomly generated task sets and realistic OpenMP programs. For each task set, we calculate the first response time bounds in (32) and (36), and then compare them with the response time bound in (5) derived by Serrano et al [4] by assuming all the tasks in the program are `untied`. In our experiments, R_0 represents the baseline response time bound in (5); R_1 represents the first response time bound in (32); R_2 represents the second response time bounds in (36). The bounds R_1 and R_2 are normalized with respect to R_0.

A. Randomly Generated Tasks

We generate the DAG G with n tasks $\{\tau_1, \cdots, \tau_n\}$, and for any task τ_j ($j \in [2, n]$), randomly assign a task T_i ($i \in [1, j-1)$) to be the parent of τ_j. We consider three types of tasks, namely, small, medium, and large tasks, with parameter ranges given in Table I. For each task, one of these types is randomly selected. Then the task parameters are chosen from the corresponding intervals with a uniform probability.

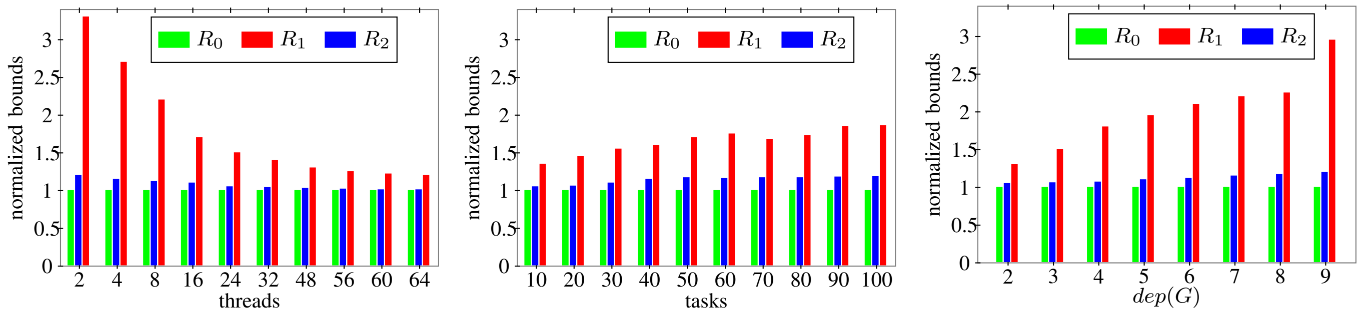
TABLE I: Task set parameters

Task Type	Small	medium	large
Vertex Number	[3,5]	[5,9]	[7,13]
Execution Time	[1,2]	[1,4]	[1,8]

For any parent task τ_i, any vertex v_{ix} of τ_i is a `taskwait` vertex with $p_{\texttt{wait}}$ probability if a predecessor of v_{ix} has an outgoing (task creation) edge of type E_2. For any task τ_i, the last vertex of τ_i points to one of its siblings that are created after τ_i through `depend` edges with $p_{\texttt{dep}}$ probability.

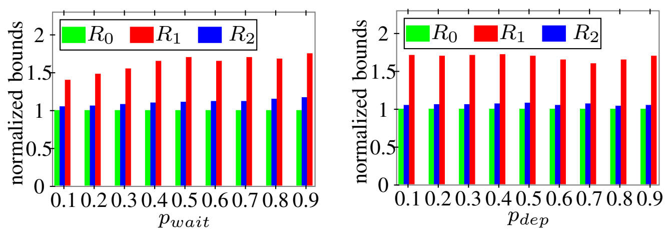
In our experiments, all the tasks are set to be `tied`. For each data point, 100 random experiments have been run.

We evaluate the response time bounds with different m (the number of threads) and different n (number of tasks), the depth $dep(G)$ and the probabilities p_{wait} and p_{dep} as shown in Fig. 7 and 8. In Fig. 7, we set $p_{wait} = p_{dep} = 0.5$, and set $m = 16$



(a) $n = 50$ (b) $m = 16$ (c) $n = 50$, $m = 16$

Fig. 7: Average bounds for random tasks ($p_{wait} = p_{dep} = 0.5$)



(a) $p_{dep} = 0.5$ (b) $p_{wait} = 0.5$

Fig. 8: Average bounds for random tasks ($n = 50$, $m = 16$)

for Fig. 7(b) and (c), and set $n = 50$ for Fig. 7(a) and (c). Moreover, we set $m = 16$ and $n = 50$ for Fig. 8, set $p_{dep} = 0.5$ for Fig. 8(a), and set $p_{wait} = 0.5$ for Fig. 8(b).

The response time bound R_2 is consistently close to the baseline bound R_0 with different parameter settings. However, the difference between R_1 and R_0 is much larger. As shown in Fig. 7a, both of the bounds R_1 and R_2 decrease as the number of threads increases. Moreover, as shown in Fig. 7c and Fig. 8a, the bounds increase as the depth $dep(G)$ and the probability p_{wait} increase, while the growth of R_2 is very slow. In Fig. 7b, the bound R_1 tends to increase as the number of tasks increases. In Fig. 8b, both of the bounds do not change significantly as the probability p_{dep} increases.

B. Realistic Benchmarks

We collect 18 OpenMP programs based on C language from several benchmarks (See in Table. II) and transform them to DAGs by a tool, called ompTG [18]. Columns 3-7 show whether the applications contain a certain structure feature, where T stands for `tied` tasks, W stands for `taskwait` clauses, D stands for `depend` clauses.

In ompTG, we parse programs by Lex & Yacc [25], which is embedded in ompi, a lightweight open source OpenMP compiler system for C programs. The output of the parser are abstract syntax trees (AST), which store useful abstract syntactic structures of the programs. We use AST to generate the DAG models which contain the basic topology information,

TABLE II: Summary of OpenMP programs in **ompTGB**.

program	benchmark	T	W	D	$dep(G)$
botsspar(br)	spec [19]	√	√	×	1
botsalgn(bn*)		√	×	×	0
poissons2D(pd*)	kastors [20]	√	×	√	0
sparseLU(su*)		√	×	√	0
strassen(sn*)		√	×	√	0
dense_algebra(da*)	dash [21]	√	×	×	0
FSM(fm*)		√	×	×	0
nbody_method(nd)		√	√	×	1
sparse_algebra(sa*)		√	×	×	0
fft(ft)	bots [22]	√	√	×	3
fib(fb)		√	√	×	9
nqueens(ns)		√	√	×	5
sort(st)		√	√	×	3
sparseLU(su)		√	√	×	1
strassen(sn)		√	√	×	2
pingpong(pg)	ompmpi [23]	√	√	×	1
overlap(op)		√	√	×	1
taskbench(th)	ompb [24]	√	√	×	1

and furthermore, measure the execution time of each vertex by executing the programs on the real hardware.

We instrument each program with instructions reading the timer at the beginning and end of each vertex, and the execution time of the vertex is the difference between the two time stamps. Although this approach cannot provide strictly safe WCET bounds, these reference WCET values give a rough idea of the workload of each vertex. In particular, currently the reference values provided by this approach are obtained on an Intel i7-4770 CPU with 3.5GHZ and 8192KB cache size, 4GB RAM size.

Some benchmarks provide the default input data for their programs. However, since these benchmarks are designed for parallel computing regions, the scale of the suggested input data is too huge for our analytical work. For example, the scale of task graph of "nqueens" exceeds 1.8G tasks when using their default input data. Therefore, we adjust the input data to keep the number of tasks generated by each program below 500.

Fig. 9 shows the response time bounds for the programs

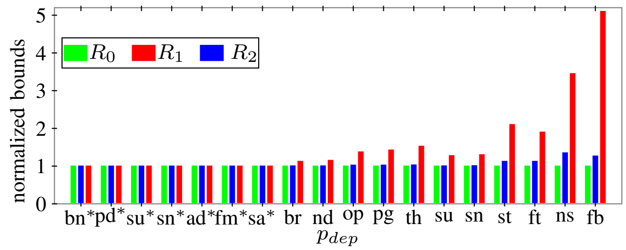


Fig. 9: Response time bounds for the benchmarks in Table. II.

in Table II by with thread number $m = 16$. (1) The first seven programs (marked with $*$) have `tied` tasks that contain no `taskwait` vertex. Each of these programs has a depth $dep(G) = 0$. (2) The next six programs (from br to sn) are non-recursive. As shown in Table II, these programs have small depths, which are 1 or 2. (3) The last four programs (from st to fb) are all from the BOTS benchmark, which contains recursive functions. The depths of these programs are large. Especially, the nested depth of fib equals 9.

As seen, the three bounds of the first seven programs (marked with $*$) are the same. For each of the next seven programs (from br to sn), the response time bounds R_1 and R_2 are very close to the baseline bound R_0. For each of the last four programs, the difference between the response time bound R_2 and R_0 remains an acceptable bound, i.e, the largest gap is no more than 0.5 (with fib. However), the difference between R_1 and R_0 becomes very large due to the large depths.

VIII. Conclusion

Multi-cores are more and more widely used in real-time systems. To fully utilize the power of multi-core processors, we must parallelize the software. OpenMP is a popular parallel programming framework in general-purpose/high-performance computing, and is also promising for real-time computing. Previous work has studied the timing analysis of OpenMP task systems, but existing techniques cannot handle `tied` tasks. In this paper we propose a new algorithm to schedule OpenMP task systems with `tied` tasks, and derive response time bounds under the new scheduling algorithm. Experiments with both realistic OpenMP programs and randomly generated workload show the effectiveness of our proposed scheduling algorithm and analysis techniques.

References

[1] Openmp application program interface version 4.5, 2015.
[2] R.E. Vargas, S. Royuela, and et.al. A lightweight openmp4 run-time for embedded systems. In *ASP-DAC*, 2016.
[3] R. Vargas, E. Quinones, and A. Marongiu. Openmp and timing predictability: A possible union? In *DATE*, 2015.
[4] M.A. Serrano, A. Melani, and et.al. Timing characterization of openmp4 tasking model. In *CASES*, 2015.
[5] D. Ferry, J. Li, and et.al. A real-time scheduling service for parallel tasks. In *RTAS*, 2013.
[6] A. Saifullah, J. Li, and et.al. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 2013.
[7] J. Li, Z. Luo, and et.al. Global edf scheduling for parallel real-time tasks. *Real-Time Systems*, 2015.
[8] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *RTSS*, 2010.
[9] A. Saifullah, D. Ferry, and et.al. Parallel real-time scheduling of dags. *IEEE Trans on PDS*, 2014.
[10] Sanjoy Baruah. Improved multiprocessor global schedulability analysis of sporadic dag task systems. In *ECRTS*, 2014.
[11] M. Qamhieh, F. Fauberteau, and et.al. Global edf scheduling of directed acyclic graphs on multiprocessor systems. In *RTNS*, 2013.
[12] M. Qamhieh, L. George, and S. Midonnet. A stretching algorithm for parallel real-time dag tasks on multiprocessor systems. In *RTNS*, 2014.
[13] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 1966.
[14] M.A Serrano, A. Melani, and et.al. Response-time analysis of dag tasks under fixed priority scheduling with limited preemptions. In *DATE*, 2016.
[15] N Guan, J.H. Sun, and et.al. Scheduling of openmp tasks. technical report, Hongkong Polytechnic University, 2016.
[16] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, 1998.
[17] G.J. Narlikar. Scheduling threads for low space requirement and good locality. *Theory of Computing Systems*, 2002.
[18] He TZ Guan N and et.al. omptg: From openmp programs to task graphs. *RTSS Workshop*, 2016.
[19] M.S. Müller, J. Baron, and et.al. Spec omp2012?xan application benchmark suite for parallel systems using openmp. In *IWOMP*, 2012.
[20] P. Virouleau, P. Brunet, and et.al. Evaluation of openmp dependent tasks with the kastors benchmark suite. In *IWOMP*, 2014.
[21] V. Gajinov, S. Stipić, and et.al. Dash: a benchmark suite for hybrid dataflow and shared memory programming models: with comparative evaluation of three hybrid dataflow models. In *ACCF*, 2014.
[22] Duran G.A., X. Teruel, and et.al. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *ICPP*, 2009.
[23] J.M. Bull, J.P. Enright, and N. Ameer. A microbenchmark suite for mixed-mode openmp/mpi. In *IWOMP*, 2009.
[24] J.M. Bull, F. Reid, and N. McDonnell. A microbenchmark suite for openmp tasks. In *IWOMP*, 2012.
[25] Xianfeng Li, Yun Liang, and et.al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM TECS*, 2008.
[26] Sanjoy Baruah and Nathan Fisher. The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. *IEEE Transactions on Computers*, 2006.

Appendix A: Computing depth $dep(\mathcal{T})$

In order to calculate the depth $dep(\mathcal{T})$, we construct a task tree $\mathcal{F} = (\mathcal{T}, \mathcal{E})$ as follows. For any tasks τ_i and τ_2 in $\mathcal{T}$, there is an edge $(\tau_i, \tau_j) \in \mathcal{E}$ if τ_j is a depending task of τ_i. Each path in $\mathcal{F}$ corresponds to a depending sequence of $\mathcal{T}$. Thus, computing $dep(\mathcal{T})$ is equally to find a path in $\mathcal{T}$ such that it contains the maximum number of `tied` non-leaves. This problem can be solved by the following recursive functions.

For any leaf $\tau_i \in \mathcal{T}$, let $\mathcal{N}_{tied}(\tau_i) = 0$, and for any non-leaf $\tau_i \in \mathcal{T}$:

$$\mathcal{N}_{tied}(\tau_i) = \begin{cases} \max\{\mathcal{N}_{tied}(\tau_j) | (\tau_i, \tau_j) \in \mathcal{E}\} + 1 & \tau_i \text{ is } \texttt{tied} \\ \max\{\mathcal{N}_{tied}(\tau_j) | (\tau_i, \tau_j) \in \mathcal{E}\} & \text{else} \end{cases}$$

where $\mathcal{N}_{tied}(\tau_i)$ denotes the maximum number of `tied` non-leaves contained in all the paths of $\mathcal{F}$ that start at τ_i. Let $dep(\mathcal{T}) = \max\{\mathcal{N}_{tied}(\tau_i) | \tau_i \in \mathcal{T}\}$, this calculation is terminated within $O(n)$ times.

Appendix B: Computing length $len(\pi_{ix})$

We know that the the last vertex of path λ_{ix} is the predecessor of a `taskwait` vertex v_{ix} of τ_i and does not travel any vertex of τ_i. From **Rules E1** and **E3.1**, the `taskwait` vertex v_{ix} of τ_i can only be connected by the vertex in τ_i or the vertex in the child of τ_i. Moreover, according to Lem. 1, there is no path that connects a non-descendant of τ_i to the descendant of τ_i and without travelling any vertex in τ_i. Thus, λ_{ix} can only travel the descendants of τ_i.

By D_{ix} we denote the subgraph consisting of the descendant τ_j of τ_i such that v_{ix} is reachable from the vertex of τ_j. More formally, D_{ix} is defined as follows.

Any task $\tau_j \in D_{ix}$ if:

- τ_j is a depending task of τ_i which joins to v_{ix}; or
- τ_j is a depending task of a task in D_{ix}.

For any τ_j and τ_l in D_{ix}, all the edges between τ_j and τ_l belong to D_{ix}.

Calculating $len(\lambda_{ix})$ is equally to find the longest path in D_{ix}. This problem is calculated by the following recursive functions.

$$\mathcal{L}(v_{jz}) = \max\{\mathcal{L}(v_{ly}) | (v_{jz}, v_{ly}) \in D_{ix}\} + C_{jz},\ \forall v_{jz} \in D_{ix}.$$

where $\mathcal{L}(v_{jz})$ denotes the length of the longest path in D_{ix} with the starting vertex v_{jz}. Let $len(\lambda_{ix}) = \max\{\mathcal{L}(v_{jz}) | v_{jz} \in D_{ix}\}$, this calculation is terminated within $O(|V|)$ times.

Appendix C: Calculation of $len^v(G)$

Given the graph G with virtual computation times, the virtual length of G can be recursively computed as follows. Denote by G_{ix} the subgraph consisting of vertices reachable from v_{ix}. The virtual length of G_{ix} is calculated by:

$$len^v(G_{ix}) = C^v_{ix} + \max\{len^v(G_{jz}) | (v_{ix}, v_{jz}) \in E\} \quad (38)$$

Finally, we calculate $len^v(G) = \max\{len^v(G_{i0}) | v_{i0}$ has no ingoing edges $\}$. This calculation procedure terminates within $O(|E|)$ times.

Appendix D: Satisfiability of Constraints

Proposition 1. *σ_{BFS} fulfils **Cons PC**.*

Proof. Under BFS, every vertex is executed after it is eligible (Lines 3 and 6). σ_{BFS} implicitly fulfils **PC** because of the definition of the eligible time in Eq.(1): for any vertex $v_{ix} \in V$, the eligible time of v_{ix}: $e_{ix} \geq f_{jz}$, $\forall (v_{jz}, v_{ix}) \in E$, which coincides with **Cons PC**. □

Proposition 2. *σ_{BFS} fulfils **Cons TSP**.*

Proof. For any $v_{ix} \in V$, we suppose that $v_{ix} \in \sigma$, and the BFS scheduler assigns v_{ix} to s_k at time t, i.e., $b_{ix} = t$. According to Eq.(4), $f_{ix} = t + C_{ix}$, and according to Eq.(3), we have $\sigma_k(t') = (i, x)$, $\forall t' \in [b_{ix}, f_{ix})$, which coincides with **Cons TSP**, and thus completes the proof. □

Proposition 3. *σ_{BFS} fulfils **Cons SE**.*

Proof. σ_{BFS} fulfils **Cons SE** if it fulfils **Cons PC** and **TSP**. This is because, in our DAG model, for any two vertices of one task, one is the predecessor of the other, and according to **Cons PC**, these two vertices should be sequentially executed. Moreover, **Cons TSP** requires each vertex of a task to be executed by one thread. In sum, a task cannot be executed by more than one thread simultaneously. Finally, according to Pro. 1 and 2, σ_{BFS} does fulfil **Cons PC** and **TSP**. It completes the proof. □

Proposition 4. *σ_{BFS} fulfils **Cons TIED**.*

Proof. Under BFS, a vertex v_{ix} of a `tied` task τ_i, with $x \leq 1$, is assigned to a thread s_k at time t, only if $\tau_i \in \Gamma_k(t)$ (Lines 3 and 14), which coincides with **Cons TIED**. □

Proposition 5. *σ_{BFS} fulfils **Cons TSC**.*

Proof. Under BFS, when assign a new `tied` task τ_i on s_k, i.e., $\sigma_k(t) = (i, 0)$, τ_i is required to be the descendant of all the tasks in $\Gamma_k(t)$ (Line 10 of Alg. 1), which coincides with **Cons TSC**. □

Appendix E: The proof of Lem. 7

Suppose that there is no vertex of key path λ_{key} is bing executed at time t.

We first show that some vertex of λ_{key} is delayed at time t. Let v_{jz} be the vertex of λ_{key} which is the latest finished at or before t, and the next vertex of λ_{key} to be executed is denoted by v_{ix}. From the definition of key path, we know that v_{jz} is the one with the latest finishing time among all the predecessors of v_{ix}. It indicates that v_{ix} of λ_{key} is eligible but delayed at time t, i.e., $b_{ix} > t \geq e_{ix}$. According to Lem. 5, v_{ix} belongs to an `untied` task, or is the first vertex of a `tied` task.

Then, the delayed vertex of λ_{key} implies that $\Gamma_k(t) \neq \emptyset$ for any idle thread s_k. Otherwise, the assignment of v_{ix} to s_k at time t does not violate **Cons E-TSC**. This implies that v_k can be executed at time t, which contradicts with the assumption. This completes the proof.