Invited Paper: Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms

Cheng-Hsiang Chiu
Department of ECE
University of Wisconsin-Madison
Madison, USA
chenghsiang.chiu@wisc.edu

Dian-Lun Lin
Department of ECE
Universiity of Wisconsin-Madison
Madison, USA
dianlun.lin@wisc.edu

Tsung-Wei Huang
Department of ECE
University of Wisconsin-Madison
Madison, USA
tsung-wei.huang@wisc.edu

***Abstract*—Many EDA applications are extremely sparse, irregular, and control-flow intensive. Parallelizing this type of application can benefit from the ability to express dynamic task parallelism across arbitrary decision-making points at runtime. Unlike the traditional construct-and-run models, dynamic task parallelism offers programmers great flexibility to parallelize EDA algorithms that incorporate complex execution logic under dynamic control flow, such as branch-and-bound techniques, on-the-fly pruning, and recursive decomposition strategies. In this paper, we introduce a new programming model that supports the dynamic building of a computational task graph. We will cover scheduling details and best practices for exploring task parallelism under dynamic control flow. We will present a real use case of our model that has successfully parallelized a static timing analysis workload.**

***Index Terms*—Dynamic task graph, task parallelism**

I. Introduction

Task graph programming (TGP) has inspired many new parallel and heterogeneous electronic design automation (EDA) algorithms [1]–[21] and large-scale machine learning problems [22]–[28]. Different from traditional loop-based models that explore parallelism across parallel loops, TGP formulates a workload as a *task graph* that models a function call as a *task* and a functional dependency as an *edge*. Figure 1(a) gives a task graph example of four tasks and four dependencies. By leveraging TGP, applications can enable top-down optimization to implement irregular parallel decomposition strategies that consist of many tasks and dependencies. Then, a TGP runtime can scale these dependent tasks across a large number of processors with dynamic load balancing [29]. As a result, the parallel computing community has yielded many successful TGP in various application domains, such as OpenMP [30], Kokkos-DAG [31], PaRSEC [32], [33], OpenCilk [34], [35], HPX [36], Taskflow [37]–[41], and Fastflow [42].

Typically, TGP is categorized to two types: *static task graph programming* (STGP) and *dynamic task graph programming* (DTGP). In STGP, applications define the task graph first and submit it to an STGP runtime for execution, as shown in Figure 1(b). Since the graph structure is known in advance, the runtime can perform whole-graph optimization. On the other hand, DTGP defines the task graph structure dynamically. Tasks and dependencies are created on the fly depending on runtime variables and control-flow results, allowing the task creation time to overlap with the task execution time (see Figure 1(c)). Thus, DTGP is often more flexible than STGP when dealing with many EDA algorithms that frequently incorporate dynamic control flow to implement irregular parallel decomposition strategies.

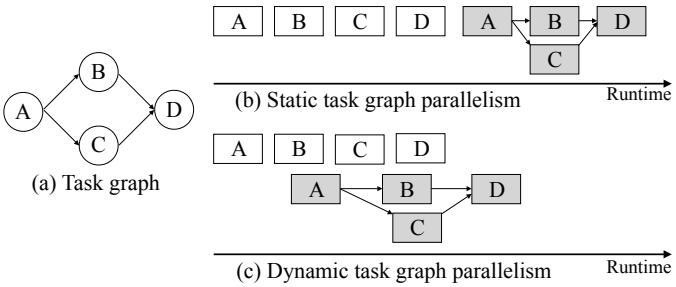

Fig. 1: An illustration of the execution diagram of a task graph. White blocks denote the task creation and gray rectangles denote the task execution. Edges refer to the dependencies. (a) A task graph. (b) The execution diagram of STGP. (c) The execution diagram of DTGP.

```
int main(){
  Executor executor;
  auto[A,fu_A]=executor.dependent_async([](){
    printf("Task A\n");
  });
  auto[B,fu_B]=executor.dependent_async([](){
    printf("Task B\n");
  }, A);
  auto[C,fu_C]=executor.dependent_async([](){
    printf("Task C\n");
  }, A);
  auto[D,fu_D]=executor.dependent_async([](){
    printf("Task D\n");
  }, B, C);
  fu_D.get();   // wait until D finishes
}
```

Listing 1: AsyncTask implementation of Figure 1(a).

In this paper, we introduce a new DTGP library called *AsyncTask* to assist EDA applications in quickly leveraging the power of DTGP. Compared to existing DTGP libraries, such as OpenMP [30] and PaRSEC [32], [33], AsyncTask

is more expressive and transparent. For example, Listing 1 demonstrates the AsyncTask code for Figure 1(a). The code *explains itself* through a clean graph description language. The program creates a task graph of four tasks, A, B, C, and D. The dependency constraints state that task A runs before task B and task C, and task D runs after task B and task C. We summarize our contributions as follows.

- *Programming Model.* We present a simple and efficient dynamic task graph programming model. Our programming model provides a clear graph description language for applications to easily and quickly describe dynamic task graph parallelism. The expressiveness of our model improves programmer's productivity when coding large and complex task graphs for EDA applications.
- *Task Scheduling Algorithm.* We present an efficient task scheduling algorithm to support our programming model. Unlike existing solutions that mostly count on heavy mutexes to schedule dependent tasks [30], [32], [33], we only use lightweight *atomic counters* to resolve dependencies between tasks, enabling a more efficient task scheduling algorithm.

We have evaluated AsyncTask on a real-world static timing analysis application. Compared with the widely-used OpenMP [30] library, AsyncTask achieves a significant speed-up of 3.41× on a large design of 420K tasks and 530K dependencies.

II. Related Works and Their Limitations

Mainstream DTGP libraries used by EDA applications include OpenMP [30], PaRSEC [32], [33], and OpenCilk [34], [35]. In this section, we discuss their implementations of Figure 1(a) and compare them with AsyncTask (Listing 1). Then, we discuss their scheduling algorithms and highlight their limitations.

A. OpenMP

OpenMP is a popular library that simplifies the development of parallel applications by adding parallelism to existing serial code through the use of compiler directives, pragmas, and runtime library routines. To implement Figure 1(a), OpenMP uses #pragma omp task construct to define a task and depend clause to specify that task's dependencies. Since OpenMP relies on a task's input and output data to describe a task's dependencies, applications need a data storage to store the data of every task. Listing 1 demonstrates the OpenMP implementation. Applications define a dynamic array dependency to store the execution results of the four tasks in the A-B-C-D order. Then applications use the entries in dependency as the inputs and outputs for a task. For example, applications use #pragma omp task to create task D and specify D's inputs to be dependency[1] (i.e., B's output) and dependency[2] (i.e., C's output), and output to be dependency[3] in the dependend clause with in or out flags. To schedule tasks, OpenMP implements a lock-based hash table, in which the key of each entry is the address of a task's input or output data, and the value of that entry is a list of tasks accessing that address. As scheduling tasks require frequent accessing and updating the hash table, the overhead of using mutexes is heavy and can impact the overall runtime performance when running large and complex task graphs with multiple threads.

```
int main(){
  int *dependency = new int [4];
  #pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp task depend(out:dependency[0])
      { printf("Task A\n"); }
      #pragma omp task depend(in: dependency[0])
                       depend(out:dependency[1])
      { printf("Task B\n"); }
      #pragma omp task depend(in: dependency[0])
                       depend(out:dependency[2])
      { printf("Task C\n"); }
      #pragma omp task depend(in: dependency[1],
                                  dependency[2])
                       depend(out:dependency[3])
      { printf("Task D\n"); }
    }
  }
  delete [] dependency;
}
```

Listing 2: OpenMP implementation of Figure 1(a).

B. PaRSEC

PaRSEC is a task-based runtime for distributed system. It leverages Domain Specific Languages (DSL) in its dataflow model to implement applications. To program a PaRSEC implementation, applications need the following steps: 1) Initialize a Message Passing Interface (MPI) engine as PaRSEC is a runtime for distributed system. 2) Define an application data structure using PaRSEC memory allocator to correctly build up the dependencies between tasks. 3) Initialize a PaRSEC taskpool to execute the tasks. 4) Define PaRSEC tasks and their function definitions. We simplify some implementation details to save more space and do our best to keep the implementation as real as possible. For instance, the original API of creating task D in Figure 1(a) needs 15 arguments and we simplify those to 8 arguments. Listing 3 implements the PaRSEC code for Figure 1(a). Applications first define a MPI engine, an application data structure dependency, and a PaRSEC taskpool. Since PaRSEC specifies dependencies through a task's input and output data as OpenMP does, the data structure dependency is used to store the computation results of the four tasks in the A-B-C-D order. Next, applications create tasks using the parsec_dtd_insert_task API. The arguments of the API include the task's function label, the task's input and output data, and an end-of-argument PARSEC_DTD_ARG_END flag. The task's inputs (flagged with INPUT) and outputs (flagged with OUTPUT) are referenced using tile_of_key together with the corresponding index in dependency. For the individual task's function definition, applications unpack the arguments in the exact order as they are specified in parsec_dtd_insert_task. For example, applications specify task D's two inputs (indexed

1 and 2 at dependency) in parsec_dtd_insert_task first and then one output (indexed 3 at dependency). Applications must obey the order to unpack them using unpack_args. To mark the end of a function definition, applications need to use the PARSEC_HOOK_RETURN_DONE flag. The task scheduling algorithm of PaRSEC is similar to OpenMP's design. Both of them rely on a lock-based hash table to manage the dependencies between tasks. The main difference is that PaRSEC additionally considers where to execute tasks that are created at a remote machine.

```
int A(parsec_task_t* this_task) {
  int *out;
  unpack_args(this_task, &out);
  printf("Task A\n");
  return PARSEC_HOOK_RETURN_DONE; }
int B(parsec_task_t* this_task) {
  int *in, *out;
  unpack_args(this_task, &in, &out);
  printf("Task B\n");
  return PARSEC_HOOK_RETURN_DONE; }
int C(parsec_task_t* this_task) {
  int *in, *out;
  unpack_args(this_task, &in, &out);
  printf("Task C\n");
  return PARSEC_HOOK_RETURN_DONE; }
int D(parsec_task_t* this_task) {
  int *in1, *in2, *out;
  unpack_args(
    this_task, &in1, &in2, &out);
  printf("Task D\n");
  return PARSEC_HOOK_RETURN_DONE; }

int main() {
  // 1. Initialize MPI
  // 2. Initialize application data, dependency
  // 3. Initialize PaRSEC taskpool, dtd_tp
  parsec_dtd_insert_task(A,
    tile_of_key(dependency,0),INPUT,
    PARSEC_DTD_ARG_END
  );
  parsec_dtd_insert_task(B,
    tile_of_key(dependency,0),INPUT,
    tile_of_key(dependency,1),OUTPUT,
    PARSEC_DTD_ARG_END
  );
  parsec_dtd_insert_task(C,
    tile_of_key(dependency,0),INPUT,
    tile_of_key(dependency,2),OUTPUT,
    PARSEC_DTD_ARG_END
  );
  parsec_dtd_insert_task(D,
    tile_of_key(dependency,1),INPUT,
    tile_of_key(dependency,2),INPUT,
    tile_of_key(dependency,3),OUTPUT,
    PARSEC_DTD_ARG_END
  );
  parsec_taskpool_wait();
}
```

Listing 3: PaRSEC implementation of Figure 1(a).

C. OpenCilk

OpenCilk is a software infrastructure for task-parallel programming. A typical OpenCilk code is to spawn threads for tasks' operations and explicitly join threads for synchronization. Listing 4 demonstrates OpenCilk implementation of Figure 1(a). Applications spawn a thread to run A using the cilk_spawn directive, and explicitly join the thread to finish A's execution using cilk_sync, ensuring the completion of A. The same pattern applies to task B, C, and D as well. As OpenCilk uses explicit synchronization directives to manage dependencies between tasks, the task scheduling algorithm is to wake up a thread from its thread pool to do a task's operation, and release that thread to the thread pool. When reaching the synchronization directives, the program execution halts until all threads finish and return to the thread pool.

```
void A(){ printf("Task A\n"); }
void B(){ printf("Task B\n"); }
void C(){ printf("Task C\n"); }
void D(){ printf("Task D\n"); }
int main() {
  cilk_spawn A();
  cilk_sync;
  cilk_spawn B();
  cilk_spawn C();
  cilk_sync;
  cilk_spawn D();
  cilk_sync;
}
```

Listing 4: OpenCilk implementation of Figure 1(a).

D. Limitations of Existing DTGP Libraries

Although OpenMP, PaRSEC, and OpenCilk have been used in many applications, we find several limitations of using them for DTGP: 1) Describing a task's dependencies through that task's input and output data is not expressive. Applications need to figure out the dataflow between two tasks to represent the task dependency, which is an indirect description and could reduce the code readability. 2) Programming a large task graph is very verbose. Applications have to explicitly indicate a task's input and output data, such as using in and out in OpenMP or INPUT and OUTPUT in PaRSEC. For PaRSEC users, they have to additionally write PARSEC_DTD_ARG_END to denote the end of input arguments and PARSEC_HOOK_RETURN_DONE to indicate the end of function definition. 3) Relying on a lock-based hash table to schedule tasks is not efficient. Their runtimes have to acquire a mutex when accessing the hash table, which introduces non-negligible lock overheads especially when running with multiple threads to schedule a complex task graph.

Because of the above limitations, we have arrived at a conclusion that we need a new DTGP library that provides an expressive programming model to simplify the building of dynamic task graph parallelism. Additionally, we need an efficient task scheduling algorithm to support the programming model without much synchronization overhead.

III. AsyncTask

At a high level, AsyncTask enables an efficient implementation of irregular parallel decomposition strategies through a top-down dynamic task graph. We provide an expressive programming model for applications to describe the task graphs easily. We also introduce a task scheduling algorithm

to support our programming model with only atomic counters to reduce the overhead of managing the dependencies between tasks.

A. Dynamic Task Graph Programming Model

To enable expressive DTGP, we directly specify a task's dependencies with its dependent tasks without describing the dependencies through the task's input and output data. Our programming model provides a simple interface dependent_async for applications to create tasks easily. Applications specify a lambda to encapsulate a task's operation followed by a list of dependent tasks in the input arguments. dependent_async will return a pair consisting of an instantiated task object and a future object which holds the execution result of that task.

Listing 1 exemplifies the code of Figure 1(a) using dependent_async. We define an executor which is a thread-safe object that manages a set of worker threads and executes our tasks. We create four tasks, A, B, C, and D. Every task defines its own lambda as the first argument followed by a list of dependent tasks. The dependent tasks must be created before the current task. For instance, task B defines its operation to print a string in the lambda and specifies one dependent task A which is created before B. Upon returning from dependent_async, we obtain a pair consisting of an instantiated task object and a future object holding the execution result of that task. After constructing all tasks, we call fu_D.get to wait for task D to finish. As we construct tasks in the order A-B-C-D, the completion of D in turns means that all the other tasks have finished their executions.

This programming model is simple and expressive. Applications only need to write a callable (or lambda) and a list of dependent tasks to create a task. There is no extra flag needed, such as in and out in OpenMP or INPUT, OUTPUT, PARSEC_DTD_ARG_END, and PARSEC_HOOK_RETURN_DONE in PaRSEC. Without these flags, our AsyncTask implementation is easy to read and debug, enabling EDA developer's high productivity when programming large and complex task graphs.

B. Algorithm

To support our programming model, we design a new task scheduling algorithm, as illustrated in Figure 2. After applications call dependent_async to create a task, the executor checks if the new task has any dependent tasks. If yes, AsyncTask builds up the dependencies between the new task and each one of its dependents (denoted with ❶), and then the new task waits until all of its dependents finish executions (denoted with ❷). Otherwise, AsyncTask directly executes the new task (denoted with ❸). Next, we dive into the three parts in more details.

❶ **Building up the dependencies**. If a new task has any dependent tasks, we need to build up the dependencies between the new task and each one of its dependent tasks. (see ❶ in Figure 2). In AsyncTask, we express every dependency with the new task presenting in the successor list of every dependent task. Assigning every task a successor list and representing dependencies using successor lists simplifies our design when it comes to resolving a dependency. For example, for the task graph in Figure 1(a), both task B and C have task A as the dependent task, and A's successor list would have two successor tasks B and C. When A finishes, AsyncTask can quickly resolve the dependencies by directly checking A's successor list rather than iterating every existing task to see what dependency to resolve. To safely and successfully add tasks into a dependent task's successor list, there are two concerns. First, when an executor adds tasks into a dependent task's successor list, we need to ensure that dependent task is alive. Since every task is an instantiated task object, it will be destroyed and returned to the operating system after it finishes the execution. To avoid adding tasks in an empty task object's successor list (see Figure 3(a)), we leverage the logic C++ smart pointer *std::shared_ptr* to retain *shared* ownership of a task between the main thread and the executor, ensuring that task remains alive throughout the entire program (see Figure 3(b)). When a worker thread finishes a task's execution, it will remove the task from the executor, decrementing the number of shared owners by one. If that counter reaches zero, the task is then destroyed.

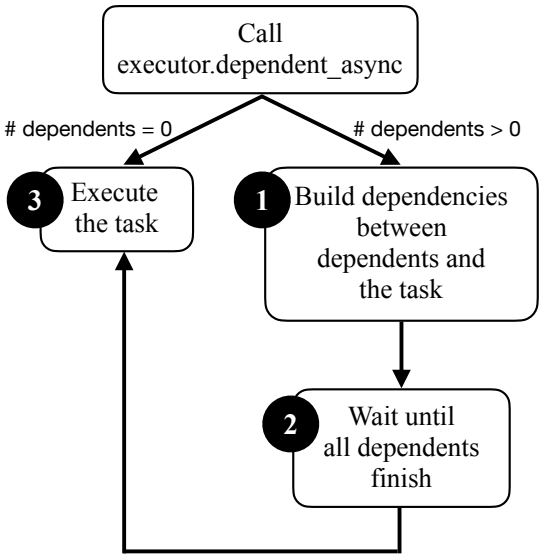

Fig. 2: A flowchart of our task scheduling algorithm.

Second, we need to protect every successor list from data race as multiple threads can add tasks in a successor list simultaneously. To avoid data race, we assign every task an atomic variable to protect its own successor list. Every atomic variable has three states FINISHED, UNFINISHED, and LOCKED. FINISHED denotes a task completion, UNFINISHED denotes an ongoing execution of a task, and LOCKED denotes that another task is adding itself to the successor list of the current task. Figure 4 visualizes how a three-state atomic variable can protect a successor list from data race. In (a), assume A has finished its execution and its

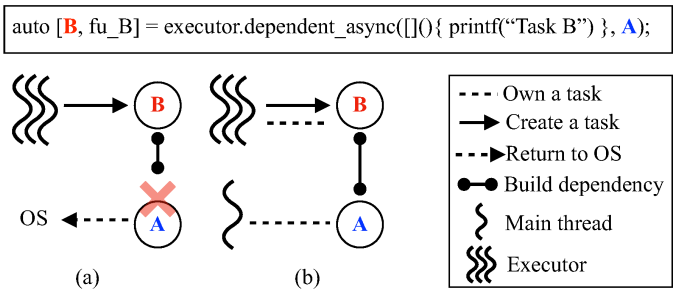

Fig. 3: An illustration of shared ownership of task A and B in Figure 1(a). (a) A finishes and returns to operating system (OS). An executor relates task B to an empty task. (b) Main thread owns A. An executor successfully relates B and A.

state is set to FINISHED. There is no need to add B and C in A's successor list. No data race on A's successor list. In (b), three tasks are performing compare-and-swap (CAS) operations on A's state at the same time. If A succeeds in this operation, then we are in the situation of (a). If B succeeds, then B changes A's to LOCKED and can add itself in A's successor list solely, as illustrated in (c). After B finishes the adding, B changes A's state back to UNFINISHED, and the whole process repeats for C.

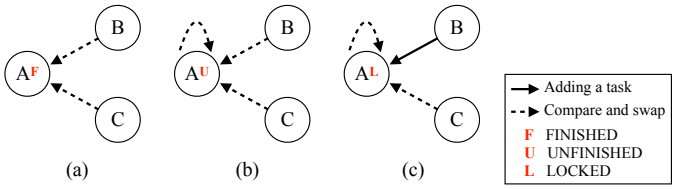

Fig. 4: An illustration of using the atomic variable to change task A's state. A, B, and C refer to the tasks in Figure 1(a). A is trying to change the state to FINISHED. B and C are trying to add themselves to A's successor list. (a) A is at the FINISHED state. (b) A is at the UNFINISHED state. (c) A is at the LOCKED state.

2 Waiting for dependent tasks to finish. After AsyncTask successfully creates a new task and builds up its dependencies, the next step for the new task is to wait for its dependent tasks to finish (see 2 in Figure 2). To achieve this, we assign every task an atomic counter to keep track of the number of its unfinished dependent tasks. The initial value is the number of dependent tasks specified in dependent_async API. When one of its dependent tasks finishes the execution, the dependent task will decrease the atomic counter of that task by one. If that atomic counter becomes zero, meaning the task has no unfinished dependent task and is ready to execute. Figure 5 visualizes the process. In (a), task B's initial atomic counter is one and it is performing the CAS operation in order to add itself in A's successor list. In (b), task B successfully added itself in A's successor list. In (c), task A finishes the execution and decreases the atomic counter of its successor B by one. Task B now has the atomic counter equal to zero and is ready to execute. We refer the atomic counter to join counter later and would use them interchangeably in the paper.

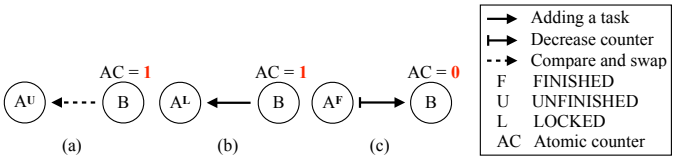

Fig. 5: An illustration of using atomic counters to represent the number of dependent tasks. A and B refer to the tasks in Figure 1(a). (a) B is performing the CAS operation on A's state. (b) B is adding itself in A's successor list. (c) A finishes its execution and decreases B's atomic counter by one.

3 Executing a task and resolve dependencies. AsyncTask executes a task when a task has no dependent task or all of its dependent tasks have finished their executions(see 3 in Figure 2). When a task finishes the execution, we need to resolve the associated dependencies between it and all of its successor tasks. To achieve this, a task iterates its successor list and decrease the atomic counter of every successor by one. Figure 6 visualizes how a task resolves the associated dependencies after it finishes the execution. In (a), task A finishes the execution, iterates its successor list, and decrease the atomic counter of B and C by one. Now, B and C have zero join counter and are ready to execute. In (b), B and C have finished the execution and both decrement D's join counter by one. D has zero join counter and is ready to execute.

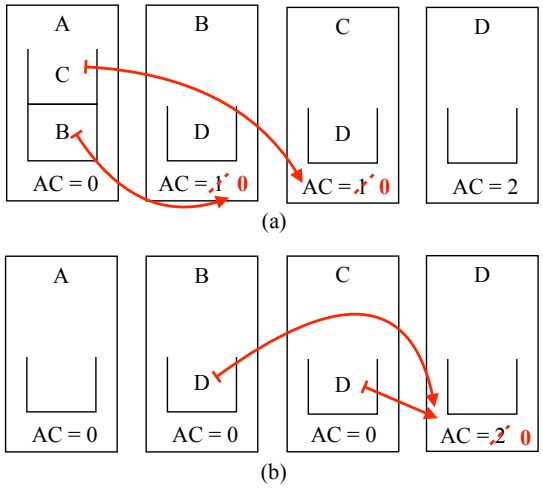

Fig. 6: An illustration of resolving dependencies after a task finishes. A, B, C, and D refer to the tasks in Figure 1(a). (a) A finishes the execution and decreases the atomic counter (AC) of B and C by one. (b) B and C finish, and both decrease D's AC by one.

C. Pseudocode

In this section, we implement the flowchart in Figure 2 based on the design overview presented in the previous

section. Algorithm 1 implements the dependent_async API which takes a callable (or lambda) and a list of dependent tasks (deps) and returns a pair consisting of the created task and a future. We first define a future object future which will hold the calculation result of the task (line 1). Then, we calculate how many dependent tasks the new task has (line 2). We initialize the new task task (line 3). Next, we iterate every dependent task and build the dependencies (lines 4:6). We build the dependencies between task and every dependent task in line 5 (details are given in Algorithm 2). Since we are in a multi-threaded environment, some dependent tasks may have finished before we build the dependencies between them and task. If all dependents finished or the task has no dependent task specified (lines 7:10), we can directly schedule task (line 8). In the end, we return a pair of the created task task and the future object future (line 10).

Algorithm 1 dependent_async(callable, deps)

```
1:  Create a future
2:  num_deps ← sizeof(deps)
3:  task ←initialize_task(callable, num_deps, future)
4:  for all dep ∈ deps do
5:      process_dependent(task, dep, num_deps)
6:  end for
7:  if num_deps == 0 then
8:      schedule_async_task(task)
9:  end if
10: return (task, future)
```

Algorithm 2 implements process_dependent in which we build the dependency between a task and one of its dependent tasks. This API takes the task task, a dependent task dep, and the number of dependent tasks num_deps in its inputs. We add task in dep's successor list if dep has not finished. First, we store the state of dep in dep_state (line 1). We create a variable target_state to store the state UNFINISHED (line 2). Next, we perform the CAS atomic operation on dep_state (line 3). If dep_state is equal to target_state (i.e. dep has not yet finished), we swap dep_state to LOCKED, which means we now enter the critical region and have the exclusive access to dep's successor list (lines 4:5). We add task in the successor list (line 4) and resume dep_state back to UNFINISHED (line 5). If dep_state is not equal to target_state, target_state would be set to dep_state atomically by the CAS operation. If target_state is set to FINISHED, that means dep has finished and we decrement task's join counter by one and update num_deps accordingly (lines 6:7). If target_state is set to LOCKED, that means some other task enters the critical section in lines 4:5 first, we reiterate to the beginning to try the whole process again (line 9).

Algorithm 3 implements the schedule_async_task API in which we execute a task, change its state to FINISHED, and then resolve the dependencies for its successors. This API takes the task task in the input. Before executing task, we change its state to FINISHED to prevent any other tasks

Algorithm 2 process_dependent(task, dep, num_deps)

```
1:  dep_state ← dep.state
2:  target_state ← UNFINISHED
3:  if dep_state.CAS(target_state, LOCKED) then
4:      dep.successors.push(task)
5:      dep_state ← UNFINISHED
6:  else if target_state == FINISHED then
7:      num_deps ←AtomDec(task.join_counter)
8:  else
9:      goto line 2
10: end if
```

from adding themselves in the task's successor list. We define target_state to be UNFINISHED (line 1). Then we perform the CAS operation on task.state. If succeed, that means task.state is equal to target_state (i.e. UNFINISHED) and is then set to FINISHED (line 2). If failed, that means some other task enters the critical section in lines 4:5 in Algorithm 2 and is adding itself in task's successor list. In such case, we reset target_state to UNFINISHED and perform the CAS operation again (line 3). When we successfully set task.state to FINISHED, we can execute task (line 5). Next, we iterate the successor list and decrement the join counter of each successor by one (lines 6:10). If any successor whose join counter becomes zero after the decrementation, we schedule that successor directly (line 8). Now, we finish executing task, we can decrement the number of task's shared owners (ref_count) by one (line 11). If task does not have any shared owner, we can delete task and return its allocated resource to the operating system (OS) (line 12).

Algorithm 3 schedule_async_task(task)

```
1:  target_state ← UNFINISHED
2:  while not task.state.CAS(target_state, FINISHED)
    do
3:      target_state ← UNFINISHED
4:  end while
5:  Invoke(task.callable)
6:  for all successor ∈ task.successors do
7:      if AtomDec(successor.join_counter) == 0 then
8:          schedule_async_task(successor)
9:      end if
10: end for
11: if AtomDec(task.ref_count) == 0 then
12:     Delete task
13: end if
```

IV. Case Study in Static Timing Analysis

We implemented AsyncTask using C++20 and evaluated its performance on an industrial static timing analysis (STA) application [1], [2] that leverages task graph parallelism to parallelize graph-based analysis (GBA). We consider the state-of-the-art open-source STA engine, OpenTimer [43], as our experimental environment. OpenTimer formulates the

TABLE I: Task ($\|V\|$) and edge ($\|E\|$) counts of three circuits.

Circuits	$\|V\|$	$\|E\|$	$\|V\| + \|E\|$
wb_dma	13125	16593	29718
tv80	17038	23087	40125
ac97_ctrl	42438	53558	95996

GBA algorithm into a task graph and schedules dependent tasks across many heterogeneous cores for parallel execution. The task graph represents the circuit graph itself and can contain millions of tasks and dependencies for large designs. Each task computes the required timing information at its corresponding node in the circuit graph (e.g., slew, delay, arrival time), while each edge represents a dependency between two tasks. Table I lists the statistics of the three circuits we use. $\|V\|$ denotes the number of the tasks in a circuit and $\|E\|$ denotes the number of the edges.

We compiled programs using Clang++17 with -std=c++20 and -O3 enabled. We ran all the experiments on a Centos Stream 8 machine with 8 Intel i7-9700K CPU at 3.60GHz and 32 GB RAM. All data is an average of ten runs. The implementation of AsyncTask is available in the Taskflow project [37].

A. Baseline

Given the large number of parallel libraries, it is impractical to compare AsyncTask with all of them. We consider OpenMP [30] as the baseline because it is a mainstream DTGP library that has been widely employed by many EDA applications. Compared with other existing DTGP libraries, OpenMP allows applications to more flexibly define task dependencies using directive-based programming (e.g., range iterator clauses).

B. Performance Comparison

Figure 7 compares the memory and runtime between AsyncTask and OpenMP with up to 16 threads for completing the analysis of the three circuits. In terms of memory usage, we see that OpenMP consistently consumes more memory than AsyncTask in all cases. This is because OpenMP implements a global lock-based hash table to track tasks and their dependencies. The key of each entry in the table is the memory address of a task's input or output data and the value is the tasks that access the corresponding memory address. The number of entries in the table grows in proportion to the edge counts. On the other hand, AsyncTask does not need a global data structure but assigns each task a successor list, which can largely reduce the overhead of lock access.

Regarding runtime performance, AsyncTask outperforms OpenMP in all cases. For example, AsyncTask is 3.19×, 3.19×, and 3.41× faster than OpenMP at 16 threads for wb_dma, tv80, and ac97_ctrl, respectively. The reason for the runtime difference comes from the design that OpenMP needs mutexes to access its global hash table in order to resolve dependencies between tasks. However, AsyncTask only uses lightweight atomic counters to resolve task dependencies.

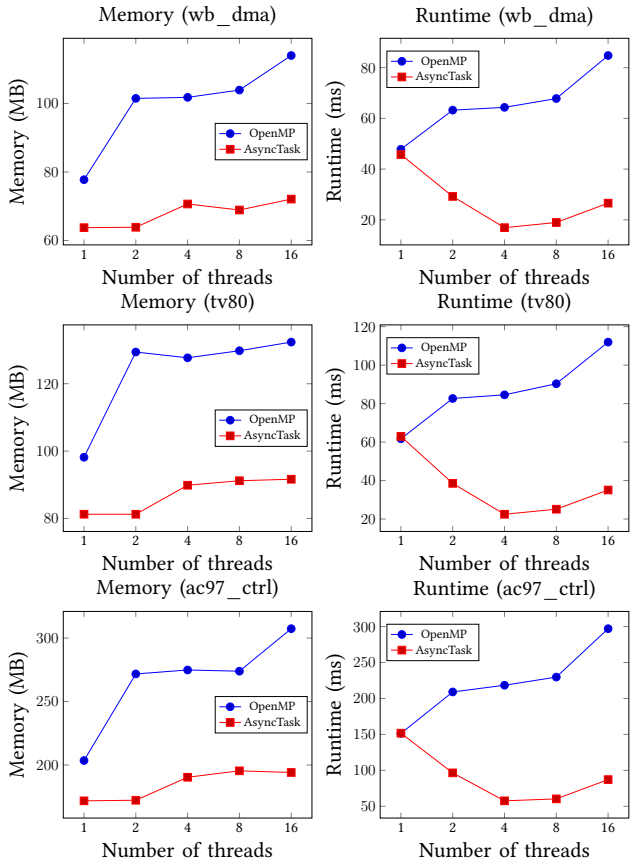

Fig. 7: Memory and runtime comparison of the STA workload on three circuits (wb_dma, tv80, ac97_ctrl) between AsyncTask and OpenMP.

V. Conclusion

In this paper, we have introduced a new DTGP library called AsyncTask to support the programming of dynamic task graph parallelism. AsyncTask has introduced a new expressive programming model supported by an efficient scheduling algorithm. We have also presented a real use case in static timing analysis and demonstrated the promising performance of AsyncTask over a mainstream DTGP library, OpenMP. AsyncTask has been integrated into the open-source Taskflow project. Future work will focus on applying AsyncTask to other EDA applications, such as distributed computing [44]–[47], macro modeling [48], and path-based analysis [49]–[53].

Acknowledgment

We are grateful for the supports of four National Science Foundation (NSF) grants, CCF-2126672, CCF-2144523 (CAREER), OAC-2209957, and TI-2229304.

References

[1] T.-W. Huang, G. Guo, C.-X. Lin, and M. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," in *IEEE TCAD*, 2021, pp. 776–789.

[2] T.-W. Huang and M. D. F. Wong, "OpenTimer: A High-Performance Timing Analysis Tool," in *IEEE/ACM ICCAD*, 2015, p. 895–902.

[3] D.-L. Lin, H. Ren, Y. Zhang, B. Khailany, and T.-W. Huang, "From rtl to cuda: A gpu acceleration flow for rtl simulation with batch stimulus," in *Proceedings of the 51st International Conference on Parallel Processing*, 2023, pp. 1–12.

[4] G. Guo, T.-W. Huang, and M. Wong, "Fast STA Graph Partitioning Framework for Multi-GPU Acceleration," in *IEEE/ACM DATE*, 2023, pp. 1–6.

[5] C.-H. Chiu and T.-W. Huang, "Composing Pipeline Parallelism Using Control Taskflow Graph," in *ACM HPDC*, 2022, p. 283–284.

[6] ——, "Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms: Late Breaking Results," in *ACM/IEEE DAC*, 2022, p. 1388–1389.

[7] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Critical Path Generation with Path Constraints," in *IEEE/ACM ICCAD*, 2021, pp. 1–9.

[8] Z. Guo, T.-W. Huang, and Y. Lin, "HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism," in *IEEE/ACM ICCAD*, 2021, pp. 1–9.

[9] ——, "A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs," in *ACM/IEEE DAC*, 2021, pp. 715–720.

[10] Y. Zamani and T.-W. Huang, "A High-Performance Heterogeneous Critical Path Analysis Framework," in *IEEE HPEC*, 2021, pp. 1–7.

[11] G. Guo, T.-W. Huang, C.-X. Lin, and M. Wong, "An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints," in *ACM/IEEE DAC*, 2020, pp. 1–6.

[12] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-Accelerated Static Timing Analysis," in *IEEE/ACM ICCAD*, 2020.

[13] Y. Lin, W. Li, J. Gu, H. Ren, B. Khailany, and D. Pan, "Abcdplace: Accelerated batch-based concurrent detailed placement on multithreaded cpus and gpus," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 5083–5096, 2020.

[14] S. Liu, P. Liao, R. Zhang, Z. Chen, W. Lv, Y. Lin, and B. Yu, "FastGR: global routing on CPU-GPU with heterogeneous task graph scheduler," *Proceedings of the 2022 Conference and Exhibition on Design, Automation and Test in Europe*, pp. 760–765, 2022.

[15] E. Dzaka, D.-L. Lin, and T.-W. Huang, "Parallel And-Inverter Graph Simulation Using a Task-graph Computing System," in *IEEE IPDPSw*, 2023, pp. 923–929.

[16] D.-L. Lin, Y. Zhang, H. Ren, S.-H. Wang, B. Khailany, and T.-W. Huang, "GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs," in *ACM/IEEE DAC*, 2023.

[17] T.-W. Huang and L. Hwang, "Task-Parallel Programming with Constrained Parallelism," in *IEEE HPEC*, 2022, pp. 1–7.

[18] T.-W. Huang, "Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs," in *IEEE HPEC*, 2022, pp. 1–2.

[19] T.-W. Huang, D.-L. Lin, Y. Lin, and C.-X. Lin, "Taskflow: A General-Purpose Parallel and Heterogeneous Task Programming System," *IEEE TCAD*, vol. 41, no. 5, pp. 1448–1452, 2022.

[20] T.-W. Huang, "qTask: Task-parallel Quantum Circuit Simulation with Incrementality," in *IEEE IPDPS*, 2023, pp. 746–756.

[21] C.-X. Lin, T.-W. Huang, T. Yu, and M. D. F. Wong, "A distributed power grid analysis framework from sequential stream graph," in *GLVLSI*, ser. GLSVLSI '18, 2018, p. 183–188.

[22] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in accelerating convolutional neural networks," in *Proceedings of the 35th International Conference on Machine Learning*, 2018, pp. 2274–2283.

[23] Z. Jia, M. Zahari, and A. Aiken, "Beyond data and model parallelism for deep neural networks," in *Proceedings of Machine Learning and Systems*, 2019, pp. 1–13.

[24] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," in *Advances in Neural Information Processing Systems*, 2019, pp. 103–112.

[25] S. Jiang, T.-W. Huang, B. Yu, and T.-Y. Ho, "SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU," in *ACM ICPP*, 2023.

[26] D.-L. Lin and T.-W. Huang, "Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism," *IEEE TPDS*, vol. 33, no. 11, pp. 3041–3052, 2022.

[27] ——, "Efficient GPU Computation Using Task Graph Parallelism," in *Euro-Par*, 2021.

[28] C.-H. Chiu, D.-L. Lin, and T.-W. Huang, "An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads," in *Euro-Par Workshop*, 2022.

[29] C.-X. Lin, T.-W. Huang, and M. D. F. Wong, "An efficient work-stealing scheduler for task dependency graph," in *IEEE ICPADS*, 2020, pp. 64–71.

[30] "OpenMP," https://www.openmp.org/.

[31] "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," in *Journal of Parallel and Distributed Computing*, 2014, pp. 3202–3216.

[32] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," in *Computing in Science Engineering*, 2013, pp. 36–45.

[33] R. Hoque, T. Herault, G. Bosilca, and J. J. Dongarra, "Dynamic task discovery in parsec: a data-flow task-based runtime," 2017, pp. 1–8.

[34] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *ACM PPOPP*, 1995, pp. 207–216.

[35] T. Schardl and I.-T. A. Lee, "OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code," in *ACM PPoPP*, 2023, pp. 189–203.

[36] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *PGAS*, 2014, pp. 1–11.

[37] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," in *IEEE TPDS*, 2022, pp. 1303–1320.

[38] T.-W. Huang, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," 2019, pp. 974–983.

[39] T.-W. Huang, "A General-Purpose Parallel and Heterogeneous Task Programming System for VLSI CAD," in *IEEE/ACM ICCAD*, 2020.

[40] T.-W. Huang, Y. Lin, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-Taskflow: A General-Purpose Parallel Task Programming System at Scale," *IEEE TCAD*, vol. 40, no. 8, pp. 1687–1700, 2021.

[41] C.-X. Lin, T.-W. Huang, G. Guo, and M. D. F. Wong, "A Modern C++ Parallel Task Programming Library," in *ACM MM*, 2019, p. 2284–2287.

[42] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *FastFlow: High-Level and Efficient Streaming on Multicore*. John Wiley and Sons, Ltd, 2017, ch. 13, pp. 261–280.

[43] "OpenTimer," https://github.com/OpenTimer/OpenTimer.

[44] T.-W. Huang, C.-X. Lin, and M. D. F. Wong, "DtCraft: A distributed execution engine for compute-intensive applications," in *IEEE/ACM ICCAD*, 2017, pp. 757–765.

[45] ——, "DtCraft: A High-Performance Distributed Execution Engine at Scale," *IEEE ICAD*, vol. 38, no. 6, pp. 1070–1083, 2019.

[46] T.-W. Huang, C.-X. Lin, G. Guo, and M. D. F. Wong, "A General-Purpose Distributed Programming System Using Data-Parallel Streams," in *ACM MM*, 2018, p. 1360–1363.

[47] T.-W. Huang, M. D. F. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran, "A distributed timing analysis framework for large designs," in *ACM/IEEE DAC*, 2016, pp. 1–6.

[48] T.-Y. Lai, T.-W. Huang, and M. D. F. Wong, "LibAbs: An Efficient and Accurate Timing Macro-Modeling Algorithm for Large Hierarchical Designs," in *ACM/IEEE DAC*, 2017.

[49] T.-W. Huang, P.-C. Wu, and M. D. F. Wong, "Fast Path-Based Timing Analysis for CPPR," in *IEEE/ACM ICCAD*, 2014, p. 596–599.

[50] ——, "UI-Timer: An Ultra-Fast Clock Network Pessimism Removal Algorithm," in *IEEE/ACM ICCAD*, 2014, p. 758–765.

[51] T.-W. Huang and M. D. F. Wong, "Accelerated path-based timing analysis with mapreduce," in *ACM ISPD*, 2015, p. 103–110.

[52] T.-w. Huang and M. D. F. Wong, "On fast timing closure: speeding up incremental path-based timing analysis with mapreduce," in *ACM/IEEE SLIP*, 2015, pp. 1–6.

[53] T.-W. Huang and M. D. F. Wong, "UI-Timer 1.0: An Ultrafast Path-Based Timing Analysis Algorithm for CPPR," *IEEE TCAD*, vol. 35, no. 11, pp. 1862–1875, 2016.