Task-Parallel Programming with Constrained Parallelism

Tsung-Wei Huang
Department of ECE, University of Utah
tsung-wei.huang@utah.edu

Leslie Hwang
Independent Researcher
leslie.k.hwang@gmail.com

***Abstract*—Task graph programming model (TGPM) has become central to a wide range of scientific computing applications because it enables top-down optimization of parallelism that governs the macro-scale performance. Existing TGPMs focus on expressing tasks and dependencies of a workload and leave the scheduling details to a library runtime. While maximizing the task concurrency is a typical scheduling goal, many applications require task parallelism to be *constrained* during the graph execution. Examples are limiting the number of worker threads in a subgraph or relating a conflict between two tasks. However, mainstream TGPMs have largely ignored this important feature of *constrained parallelism* in a task graph. Users have no choice but to implement a separate and often sophisticated scheduling solution that is neither generalizable nor scalable. In this paper, we propose a *semaphore* programming model and a scheduling method both of which can be easily integrated into an existing TGPM to support constrained parallelism. We have demonstrated the effectiveness and efficiency of our approach in real applications. As an example, our semaphore model speeds up an industrial circuit placement workload up to 28%.**

I. Introduction

Task-based parallel computing system plays an essential role in advanced scientific computing [43]. Unlike loop-based models, task-parallel systems typically leverage a *task graph programming model* (TGPM) to encapsulate function calls and their dependencies in a top-down *task graph*. TGPM is advantageous in dealing with irregular parallel decomposition strategies and scaling them to a large number of processors comprising manycore central processing units (CPUs) and graphics processing units (GPUs). As a result, a great deal amount of TGPM research has been proposed in recent years. A representative but incomplete list of work includes Taskflow [26], oneTBB FlowGraph [1], StarPU [11], Legion [12], Kokkos-DAG [19], PaRSEC [13], HPX [30], and Fastflow [9]. These systems have enabled vast success in a variety of scientific computing applications, such as machine learning, data analytics, and simulation.

When programmers describe a workload in a task graph, the system runtime takes care of all the scheduling details, including load balancing and concurrency control. While maximizing the task concurrency is a main goal to fast completion time, many applications require *constrained task parallelism* inside a task graph. For example, an embedded application task graph may only allow a fixed number of workers to enter a subgraph of restricted computing resources; a circuit routing workload may not invoke two workers to compute two regions that have a conflict. Unfortunately, existing TGPMs have largely ignored this type of constrained task parallelism. As a result, users resort to a different scheduling heuristic or a client-side partition algorithm, both resulting in rather sophisticated implementation. Also, this workaround makes it challenging to take advantage of important features that have been established in an existing runtime, such as dynamic load balancing that is known difficult to program correctly.

Consequently, in this paper we introduce a lightweight programming model and scheduling method to overcome the challenge of constrained parallelism in a task graph. Our model is very general and can handle common task parallelism constraints, including counting semaphores, limiting subgraph concurrency, and resolving conflict parallelism. More importantly, our approach can be easily integrated into an existing TGPM and its scheduling runtime, taking advantage of all established library features. We have evaluated the effectiveness of our approach atop two TGPMs, Taskflow [5] and oneTBB FlowGraph [1], and demonstrated its efficiency on real applications. As an example, our semaphore model speeds up an industrial circuit placement workload up to 28%.

II. Motivation: Parallel CAD Algorithms

This research is motivated by our parallel computing projects on high-performance computer-aided design (CAD) algorithms. CAD is a software method to help people design integrated circuits (ICs) or very-large-scale integration (VLSI) systems [47]. It takes a high-level hardware description language (HDL) of an electronic component and run design automation algorithms (e.g., placement, routing) to generate a physical layout. As the design complexity continues to increase, new CAD algorithms must harness the power of parallelism to reduce the ever-increasing runtime [42]. However, parallelizing CAD is an extremely challenging job as it involves many special constraints that are rarely addressed by existing high-performance computing (HPC) systems.

Consider an abstract task graph in Figure 1. The task graph (partially) models a detailed placement algorithm to optimize cell locations in a chip using minimal wirelength [40]. The middle four tasks call an algebra library to solve a linear system. From the algorithm standpoint, the four solver tasks can run in parallel. In practice, however, the algebra library only allows a certain amount of parallelism, such as two worker threads to enter the loop, as a result of license and

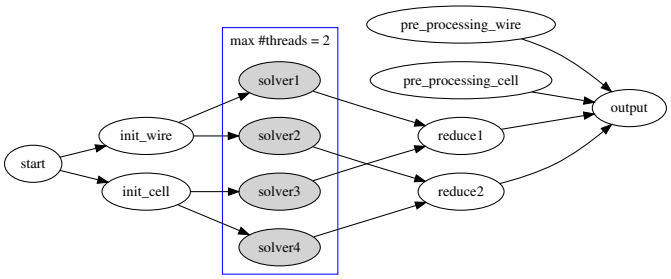

Fig. 1: A task graph of a circuit placement algorithm that restricts two worker threads to run the four solver tasks.

implementation restrictions. A common solution adopted by CAD developers is to partition the constrained region out of the task graph and execute it with two threads. This solution is simple, but it requires running an expensive partitioning algorithm and adding an additional synchronization mechanism before and after the partition. Also, this organization can potentially degrade the performance because tasks like `reduce1` and `reduce2` should have immediately started whenever their dependencies are met.

Another example is the conflict graph parallelism, which frequently happens in many parallel physical design algorithms, such as routing [32], [41]. Consider the example in Figure 2. The left task graph represents a routing algorithm and the right undirected graph imposes four conflict constraints among four routing tasks, `route_B`, `route_C`, `route_E`, and `route_F`. Individual routing tasks can run in parallel as long as their routing regions (e.g., bounding box of a net) do not overlap. In this example, `route_B` and `route_C` cannot run simultaneously as there is a conflict between `net_B` and `net_C`; same for `route_B` and `route_F` and so on. A common solution adopted by CAD developers is to transform the conflict graph to a *directed acyclic graph* (DAG) and induce additional task dependencies in the task graph. For instance, we can induce a direction from `net_B` to `net_F`, which results in a task dependency from `route_B` to `route_F`. However, this solution is static and often require expensive decision algorithms, such as finding a maximum independent set (MIS), to maximize the parallelism [32].

After years of research, we have arrived at the key conclusion that there is a need for new programming model and scheduling algorithm to handle constrained parallelism in a task graph. Instead of redesigning everything from scratch, which is not practical, we favor a drop-in solution that can be easily integrated into an existing task graph computing system.

III. Semaphore Programming Model

For the purpose of demonstration, we explain our programming model for constrained parallelism atop the popular open-source software, Taskflow [26]. Taskflow is a general-purpose parallel and heterogeneous task programming system. It introduces a new *control taskflow graph* (CTFG) programming model that enables end-to-end expressions of dependent tasks with in-graph control flow. A simple CTFG program is shown in Listing 1. The code explains itself.

```
tf::Taskflow taskflow;
tf::Executor executor;
auto [A, B, C, D] = taskflow.emplace(
  [] () { std::cout << "Task A"; },
  [] () { std::cout << "Task B"; },
  [] () { std::cout << "Task C"; },
  [] () { std::cout << "Task D"; }
);
A.precede(B, C);  // A runs before B and C
D.succeed(B, C);  // D runs after  B and C
executor.run(tf).wait();
```

Listing 1: A task graph of four tasks and four task dependencies.

```
tf::Executor executor;
tf::Taskflow taskflow;
// creates a semaphore with the counter = 2
tf::Semaphore semaphore(2);
std::vector<tf::Task> tasks {
  taskflow.emplace([](){ std::cout << "A"; }),
  taskflow.emplace([](){ std::cout << "B"; }),
  taskflow.emplace([](){ std::cout << "C"; }),
  taskflow.emplace([](){ std::cout << "D"; }),
  taskflow.emplace([](){ std::cout << "E"; })
};
// constrains 2 workers to run 5 parallel tasks
for(auto task : tasks) {
  task.acquire(semaphore);
  task.release(semaphore);
}
executor.run(taskflow).wait();
```

Listing 2: Our semaphore programming model.

Listing 2 shows our programming model for constrained parallelism, which leverages the concept of *semaphore* to limit the maximum concurrency in a section or subgraph of tasks. Here, it lists five parallel tasks. Under normal circumstances, these five tasks can be executed simultaneously. However, we create a semaphore with an initial count `2`, and have each task acquire and release that semaphore before and after executing its work, respectively. This semaphore limits the number of concurrently running tasks to only two. Figure 3 shows one possible execution result of Listing 2.

```
tf::Semaphore semaphore(1);
int counter = 0;  // non-atomic integer counter
for(int i=0; i<6; i++) {
  tf::Task f = taskflow.emplace(
    [&](){ counter++; }
  ).name(
    "from-"s + std::to_string(i)
  );
  tf::Task t = taskflow.emplace(
    [&](){ counter++; }
  ).name(
    "to-"s + std::to_string(i)
  );
  f.precede(t);
  f.acquire(semaphore);
  t.release(semaphore);
}
executor.run(taskflow).wait();
assert(counter == 12);
```

Listing 3: Sequential execution of Figure 4 using a binary semaphore.

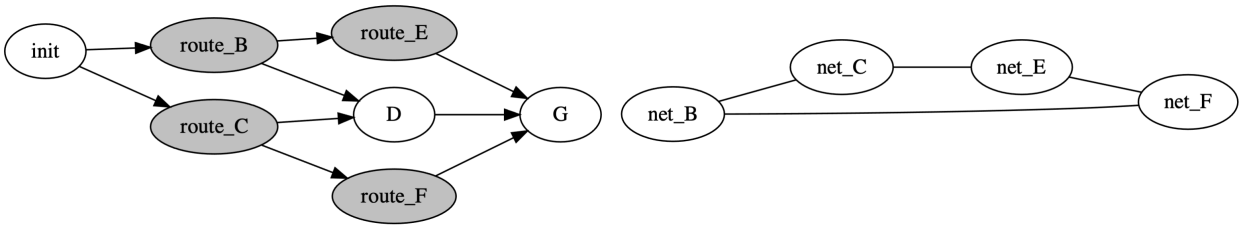

Fig. 2: A task graph (left) of a circuit routing algorithm that incorporates four conflict constraints (right).

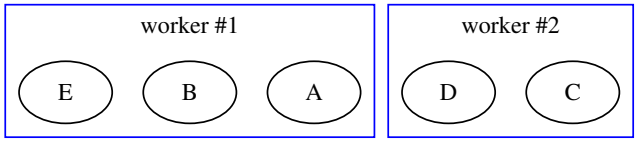

Fig. 3: One possible execution result of Listing 2. The semaphore restricts at most two workers to run five independent tasks.

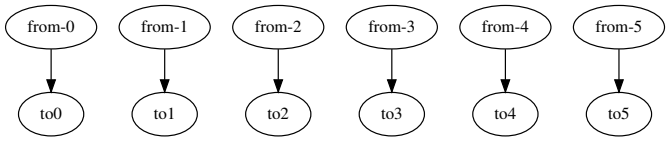

Fig. 4: A task graph of six parallel group pairs.

Our semaphore model is very flexible. A task can acquire and release a semaphore, or just acquire or just release it. Acquiring and releasing a semaphore can occur across different taskflows, as long as the two operation counts equal each other. For example, Listing 3 creates a binary semaphore to sequentially run six parallel groups of two linearly dependent task. Without semaphores, each group (e.g., `from-0` $\rightarrow$ `to-0`), can run simultaneously. However, the program asks each "from" task to acquire the semaphore before running its work and not to release it until its "to" task is done. This constraint forces each pair of tasks to run sequentially. When the program finishes, we can safely assert the non-atomic counter to be `12`.

```
tf::Semaphore BC(1), CE(1), EF(1), BF(1);
route_B.acquire(BC).release(BC);
route_C.acquire(BC).release(BC);
route_C.acquire(CE).release(CE);
route_E.acquire(CE).release(CE);
route_E.acquire(EF).release(EF);
route_F.acquire(EF).release(EF);
route_B.acquire(BF).release(BF);
route_F.acquire(BF).release(BF);
```

Listing 4: Implementation of the conflict task parallelism in Figure 2.

Another important application of our semaphore is the expression of conflict task parallelism. Conflict task parallelism is a very frequently occurring constraint in CAD algorithms because nets that are connected to each other introduce computational task dependencies. Listing 4 implements the conflict graph in Figure 2 using four binary semaphores, each representing a conflict edge. For each conflict edge, we have both of its tasks acquire and release the corresponding semaphore. For example, `route_C` will acquire and release `BC` and `CE`, to avoid the conflict with `route_B` and `route_E`. Since the semaphore counts on `1`, only one task of the edge will run at a time.

IV. Semaphore Design and Scheduling

At a high level, the design and scheduling of our semaphore model is as follows: A `tf::Semaphore` object starts with an initial count. As long as the count is above zero, tasks can acquire the semaphore and do their work. If the count becomes zero, the task trying to acquire the semaphore will not run its work but goes to a waiting list of that semaphore. That is, each semaphore has a vector to keep track of which task is waiting on it. When the semaphore is released by a task, it informs the scheduler to schedule tasks from the waiting list. Since a semaphore may be accessed by multiple tasks (and hence worker threads), we protect all semaphore operations using a lock.

Algorithm 1: Acquire and release algorithms in `tf::Semaphore`

Per semaphore: $counter$: internal counter
Per semaphore: $mutex$: lock to protect the counter
Per semaphore: $waiters$: vector of tasks waiting on this semaphore

1 **Semaphore** `acquire(`$task$`)`**:**
2 scoped_lock($mutex$);
3 **if** $counter > 0$ **then**
4 $counter \leftarrow counter - 1$;
5 **return true**;
6 **end**
7 $waiters \leftarrow waiters \cup task$;
8 **return false**;
9
10 **Semaphore** `release()`**:**
11 scoped_lock($mutex$);
12 $counter \leftarrow counter + 1$;
13 **return** move($waiters$);

Algorithm 1 shows the acquire and release methods in `tf::Semaphore`. Unlike the typical semaphore implemen-

tation, both methods are designed to work with a generic task graph scheduler in a wait-free fashion (except the lock protection). When a task fails to acquire the semaphore, it joins the waiting list and returns `false` to inform the scheduler not to run its work. Each time a task releases a semaphore, it will pull out all waiting tasks and return them to the scheduler. Notice that the release method always returns waiting tasks (using the cheap move semantic) regardless of its counter value, because the scheduler can reschedule at least one task from the waiting list of this semaphore.

Algorithm 2: Acquire and release algorithms in a library task

```
   Per task: S+: the set of semaphores to acquire
   Per task: S−: the set of semaphores to release
 1 Task acquire_all(tasks):
 2     A ← null;
 3     foreach s ∈ S+ do
 4         if s.acquire(this) = false then
 5             foreach a ∈ A do
 6                 tasks ← tasks ∪ a.release();
 7                 return false;
 8             end
 9         end
10         A ← A ∪ s;
11     end
12     return true;
13
14 Task release_all():
15     R ← null;
16     foreach s ∈ S− do
17         R ← R ∪ s.release();
18     end
19     return R;
```

Based on the semaphore design in Algorithm 1, we can extend the task interface of most existing task graph libraries to include two methods, `acquire_all` and `release_all`, as shown in Algorithm 2. The two methods will be called by the scheduler to acquire and release all semaphores that are assigned to a task. The first method, `acquire_all`, tries to acquire all semaphores in S^+. If one semaphore fails, it releases all acquired semaphores so far and returns their waiting tasks to the scheduler. The second method, `release_all`, simply releases all semaphores in S^- and returns all their waiting tasks to the scheduler.

A key advantage of Algorithm 2 is its drop-in integration with existing task graph schedulers. Most task graph schedulers define a pluggable *prologue-epilogue* interface that allows developers to run custom functions before and after the execution of a task. A common use case is adding observers to track thread activities during the execution of a task graph, i.e., which worker thread is running which task. As a result, the acquire and release guards for scheduling a task with its semaphores can be efficiently implemented as in Algorithm 3.

Algorithm 3: Acquire and release algorithms in a library scheduler

```
 1 Scheduler prologue_acquire(task):
 2     if task.S+ ≠ null then
 3         A ← null;
 4         if task.acquire_all(A) = false then
 5             schedule(A);
 6             signal the scheduler to return;
 7         end
 8     end
 9
10 Scheduler epilogue_release(task):
11     if task.S− ≠ null then
12         R ← task.release_all();
13         schedule(R);
14     end
```

The algorithm is self-explanatory.

While it is possible to further optimize Algorithms 1–3, such as pooling semaphores in a scheduler for the given assignment and reducing contention through fine-grained task control, we find these optimization strategies often too intrusive to the library runtime. Also, our application experience leads us to believe that the proposed semaphore model is simple and efficient enough for most applications wihout sophisticated semaphore usage.

V. Experimental Results

We implemented our semaphore programming model and scheduling algorithm atop two state-of-the-art task graph computing systems, Taskflow [26] and oneTBB FlowGraph [1]. The proposed semaphore model is very lightweight, increasing fewer than 500 lines of code in either of the systems. We evaluated the effectiveness of our design on two real CAD workloads, (1) a placement workload that limits the maximum concurrency in subgraphs of tasks (e.g., Figure 1) and (2) a routing workload that imposes a conflict graph on a task graph (e.g., Figure 2). We compared the performance between two implementation results: one with our semaphore model and one with existing heuristics commonly adopted by CAD developers. All experiments ran on a Ubuntu Linux 5.0.0-21-generic x86 64-bit machine with 40 Intel Xeon Gold 6138 CPU cores at 2.00 GHz and 256 GB RAM. We compiled all programs using GNU GCC-9.2.1 with C++17 standards and optimization flags `-O2` enabled. All data is obtained from an average of ten runs.

A. Placement Algorithm with Constrained Parallelism

Placement is a critical step in layout optimization. The goal is to optimize the interconnect among millions of logic gates or instances to improve timing and power. Connected instances are grouped to a *net* with interconnect modeled in Manhattan distance. An example is shown in Figure 5. We implement a detailed placement algorithm in ABCDPlace [40]

that formulates a local reordering algorithm in a parallel task graph to optimize the interconnect. The task graph restricts several sections of tasks to have no more than four or eight workers calling an internal linear system solver. To handle this concurrency constraint, we assign each section a semaphore with the internal counter initialized to four or eight. Our baseline implements the same algorithm but partitions the task graph into several subgraphs around sections of constrained parallelism and execute each partition using a different number of workers. The baseline implementation essentially reproduces the heuristic used in ABCDPlace [40] and other existing placement engines.

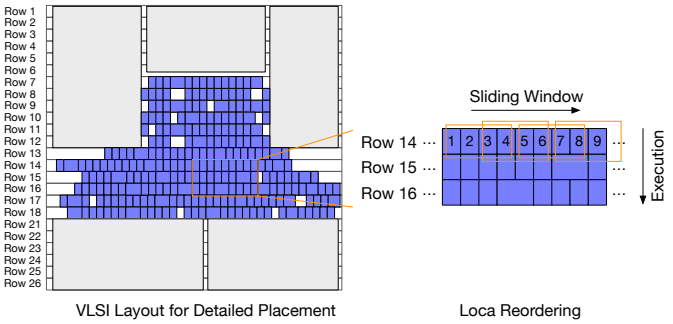

Fig. 5: Example of a VLSI layout for placement (left) and the execution of the local reordering algorithm to optimize the interconnect among gates (right).

Table I shows the benchmark statistics and the performance results of baseline and our semaphore design atop Taskflow and oneTBB. The numbers of tasks, dependencies, and semaphores in the placement task graph are denoted as $|V|$, $|E|$, and $|S|$, respectively. The baseline consists of the graph partition time and the task graph execution time, while both Taskflow and TBB directly solve the problem using the proposed semaphore. The first four small circuits (adaptec) use four semaphores, and the rest four large circuits (bigblue) use eight semaphores. All these eight circuits are real industrial designs [44]. The largest design, bigblue4, has over 2.1M gates and 2.2M nets, and its task graph has over 8K tasks and 15K dependencies.

In terms of runtime performance, our semaphore with Taskflow achieves 18–26% speed-up over the baseline. The result is slightly faster than oneTBB in most cases. We attribute this to the scheduler design of Taskflow [26]. However, our goal is not to compare different task graph systems but focus on the advantage semaphore brings. With semaphore, there is no need to partition the graph, which takes about 8–13% of the total runtime; for instance, bigblue4 spends 12% of the total time on partition. More importantly, our semaphore model allows the entire workload to run in an end-to-end task graph. The scheduler can more efficiently handle task overlap and load balancing than the baseline.

B. Routing Algorithm with Conflict Parallelism

Routing is another key step in layout optimization after placement. The goal is to establish the actual routes of wires among placed gates using minimal wirelength. Routing is an extremely time-consuming step (e.g., hours to complete a million-gate design). Hence, there have been many parallel routing algorithms in recent years to reduce the long runtime [32]. Unlike placement, parallel routing algorithm (routing task graph) needs to deal with conflict parallelism induced by overlapped regions of nets. Figure 6 gives an example of a routing conflict graph induced by net overlaps. This type of conflict parallelism can be easily expressed by our semaphore model without changing the routing task graph. Without semaphore, a common heuristic is to iteratively find an MIS in the conflict graph and induce linear dependencies between iterations onwards. We considered the state-of-the-art FastGR [41] that implements this heuristic as our baseline.

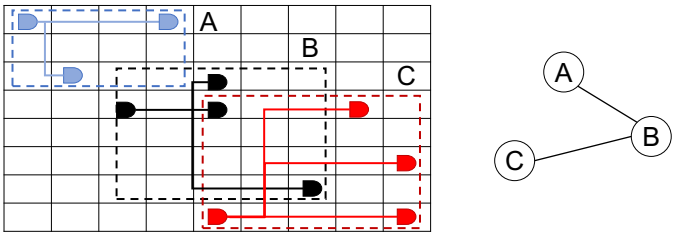

Fig. 6: Example of a routing problem (left) and its conflict graph (right) induced by net bounding-box overlaps.

Table II shows the benchmark statistics and the runtimes of baseline and our semaphore implementation atop Taskflow and oneTBB. We used six real circuit designs from ICCAD 2019 Contest [45]. The largest design, 19test9, has over 895K nets to route in a 1337×1433 routing grid, resulting in a task graph of 895K tasks and 24 dependencies. Notice that the dependency count is much fewer than the task count because we use semaphores to capture the conflict graph induced by nets. Similar to placement, the baseline consists of the time to find MISs in all iterations and the time to run the routing task graph.

In terms of runtime performance, our semaphore with Taskflow is 13–28% faster than the baseline. The implementation atop Taskflow is slightly faster than oneTBB, due to the scheduling efficiency of Taskflow [26]. The independent set decision problem is known to be NP-complete. Even with the MIS heuristic in FastGR [41], it still accounts for 6–14% of the total runtime. This overhead, however, can be totally avoided by our semaphore model that directly solves the problem at the programming level. For all circuits, the cost of creating semaphores is less than 1 second, which is acceptable. We believe our solution can inspire more efficient parallel algorithms for other CAD problems that exhibit similar characteristics of conflict parallelism [17], [21]–[24], [27]–[29], [39].

C. Other Use Cases

In addition to the previous two VLSI applications, we want to highlight other real use cases of our semaphore model. Before we put together our technical innovations in this writing, we had integrated our semaphore model into Taskflow [26]

TABLE I: Runtime Performance of Placement Workload

circuit	#gates	#nets	$\|V\|$	$\|E\|$	$\|S\|$	Baseline			Taskflow (ms)	TBB (ms)
						Partition (ms)	Graph (ms)	Total (ms)		
adaptec1	211447	221142	5240	10435	4	108.3	1041.3	1149.6	890.3	900.4
adaptec2	255023	266009	5240	10435	4	140.2	1168.7	1308.9	1065.2	1103.4
adaptec3	451650	466758	5240	10435	4	210.6	1949.6	2157.2	1766.8	1832.1
adaptec4	496045	515951	5240	10435	4	214.3	1978.7	2193	1708.4	1792.5
bigblue1	278164	284479	8430	15887	8	141.2	915.4	1056.6	778.5	770.4
bigblue2	557866	577235	8430	15887	8	222.5	2069.8	2292.3	1808.5	1743.5
bigblue3	1096812	1123170	8430	15887	8	411.0	4665.2	5076.2	4029.9	4198.4
bigblue4	2177353	2229886	8430	15887	8	793.2	5367.5	6160.7	4649.7	4834.8

TABLE II: Runtime Performance of Routing Workload

circuit	#nets	Grid Graph	$\|V\|$	$\|E\|$	$\|S\|$	Baseline			Taskflow (s)	TBB (s)
						MIS (s)	Graph (s)	Total (s)		
18test5	72394	619×613	72426	24	71985	17.7	113.1	130.8	94.4	104.4
18test8	179863	619×613	179895	24	191987	35.2	304.1	339.3	274.8	275.2
18test10	182000	606×522	182032	24	290386	48.8	406.0	454.8	394.2	398.4
19test7	358720	1053×1011	358752	24	359746	74.5	678.5	753	615.9	631.1
19test8	537577	1202×1138	537609	24	539611	34.4	501.8	536.2	403.2	441.7
19test9	895252	1337×1433	895284	24	899341	90.1	703.4	793.5	587.1	600.3

and tested it rigorously. The feedback is encouraging. For instance, we have collaborated with a company (anonymous due to intellectual property) to deploy our semaphore interface in their embedded vision applications for improving energy efficiency. Additionally, we have helped a robotic vendor design their motion planning algorithms and incorporate constrained parallelism in resource-critical sections.

VI. Related Work

Through the evaluation of parallel programming standards, *task-based* execution model has been proven to scale well with the future HPC systems [43]. Directive-based programming models [2]–[4], [20] allow users to augment program information with task and dependency rules (e.g., OpenMP dependency clauses) and delegate the code generation to compilers. These models are good at static graph construction but cannot easily handle dynamic scenarios where graph structure depends on runtime variables. Functional approaches [1], [9], [11]–[13], [16], [19], [25], [26], [30], [31], [33], [34], [36]–[38] offer either implicit or explicit task graph constructs that are more flexible in runtime control and on-demand tasking. However, most of these systems focus on modeling and scheduling the task graph itself, and they do not anticipate constrained parallelism which is essential for important computer engineering applications, such as high-performance CAD algorithms.

Existing task graph schedulers have largely adopted *work stealing* to achieve dynamic load balancing [10]. Depending on the applications, work-stealing algorithms have many variants. A list of representative but incomplete algorithms include general-purpose work-stealing loops [1], [26], balanced worker management [18], adaptive stealing strategy [8], [35], deterministic design [46], and data locality-aware runtime [15], [49]. These algorithms typically define a pluggable interface between task-level scheduling and work-level management, allowing us to easily integrate our semaphore model into their runtimes without modifying their work-stealing algorithms.

Scheduling parallel tasks under constrained parallelism has been addressed in several applications, such as makespan minimization with machine conflicts [14], conflict graph-based concurrent transmission scheduling for wireless networks [48], conflict open-shop scheduling in operation research [6], knapsack programming with conflict graph [7], net dependency breaking in CAD [32], [40], [41], and so on. While these applications have solved constrained parallelism in certain aspects, their solutions have been application-specific. There are no general-purpose solutions for programming task graphs with constrained parallelism.

VII. Conclusion

In this paper, we have introduced a semaphore-based programming model and scheduling algorithm to handle constrained parallelism in a task graph. Our model is very general and can handle common task parallelism constraints, such as limiting the maximum concurrency of a subgraph and resolving conflict task parallelism. More importantly, our semaphore model is lightweight and can be easily integrated into an existing task graph computing system to reuse all its infrastructure, in particular, dynamic load balancing. We have evaluated the effectiveness of our approach atop two state-of-the-art systems, Taskflow and oneTBB FlowGraph, and demonstrated its efficiency in two real design automation applications. As an example, our semaphore model speeds up an industrial circuit placement workload up to 28%.

Acknowledgment

We are grateful for the support of National Science Foundation (NSF) grants, CCF-2126672, CCF-2144523 (CAREER), and OAC-2209957. Special thanks go to Taskflow users for providing us feedback to improve our semaphore design.

REFERENCES

[1] Intel oneTBB. https://github.com/oneapi-src/oneTBB.

[2] OmpSs. https://pm.bsc.es/ompss.

[3] OpenACC. http://www.openacc-standard.org.

[4] OpenMP. https://www.openmp.org/.

[5] Taskflow. https://taskflow.github.io/.

[6] Open shop scheduling problems with conflict graphs. *Discrete Applied Mathematics*, 227:103–120, 2017.

[7] A fast algorithm for knapsack problem with conflict graph. *Asia-Pacific Journal of Operational Research*, 38, 2021.

[8] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive Work Stealing with Parallelism Feedback. In *PPoPP*, pages 112–120, 2007.

[9] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. John Wiley and Sons, Ltd, 2017.

[10] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *ACM SPAA*, pages 119–129, 1998.

[11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, 2011.

[12] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *IEEE/ACM SC*, pages 1–11, 2012.

[13] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science Engineering*, 15(6):36–45, 2013.

[14] Moritz Buchem, Linda Kleist, and Daniel Schmidt genannt Waldschmidt. Scheduling with Machine Conflicts, 2021.

[15] Quan Chen, Minyi Guo, and Haibing Guan. LAWS: Locality-Aware Work-Stealing for Multi-Socket Multi-Core Architectures. In *ACM ICS*, page 3–12, 2014.

[16] Cheng-Hsiang Chiu and Tsung-Wei Huang. Composing Pipeline Parallelism Using Control Taskflow Graph. In *ACM HPDC*, pages 283—284, 2022.

[17] Cheng-Hsiang Chiu and Tsung-Wei Huang. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE Design Automation Conference (DAC)*, 2022.

[18] Xiaoning Ding, Kaibo Wang, Phillip B. Gibbons, and Xiaodong Zhang. BWS: Balanced Work Stealing for Time-sharing Multicores. In *ACM EuroSys*, pages 365–378, 2012.

[19] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014.

[20] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *IEEE IPDPS*, pages 1299–1308, 2013.

[21] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. GPU-accelerated Pash-based Timing Analysis. In *ACM/IEEE Design Automation Conference (DAC)*, 2021.

[22] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. GPU-accelerated Static Timing Analysis. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 1–8, 2020.

[23] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[24] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin Wong. OpenTimer 2.0: A New Parallel Incremental Timing Analysis Engine. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 40(4):776–789, 2021.

[25] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE IPDPS*, pages 974–983, 2019.

[26] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, volume 33, pages 1303 – 1320, 2022.

[27] Tsung-Wei Huang and Martin Wong. OpenTimer: A high-performance timing analysis tool. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 895–902, 2015.

[28] Tsung-Wei Huang and Martin Wong. UI-Timer 1.0: An Ultrafast Path-Based Timing Analysis Algorithm for CPPR. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 35(11):1862–1875, 2016.

[29] Tsung-Wei Huang, Martin D. F. Wong, Debjit Sinha, Kerim Kalafala, and Natesan Venkateswaran. A distributed timing analysis framework for large designs. In *ACM/IEEE Design Automation Conference (DAC)*, pages 116:1–116:6, 2016.

[30] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A Task Based Programming Model in a Global Address Space. In *PGAS*, pages 6:1–6:11, 2014.

[31] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A Portable Concurrent Object Oriented System Based on C++. In *ACM ASPLOS*, page 91–108, New York, NY, USA, 1993.

[32] Haocheng Li, Gengjie Chen, Bentian Jiang, Jingsong Chen, and Evangeline F. Y. Young. Dr. cu 2.0: A scalable detailed routing framework with correct-by-construction design rule satisfaction. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 1–7, 2019.

[33] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin D. F. Wong. An Efficient and Composable Parallel Task Programming Library. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2019.

[34] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin D. F. Wong. A modern c++ parallel task programming library. In *ACM Multimedia Conference*, page 2284–2287, 2019.

[35] Chun-Xun Lin, Tsung-Wei Huang, and Martin D. F. Wong. An efficient work-stealing scheduler for task dependency graph. In *IEEE ICPADS*, pages 64–71, 2020.

[36] Dian-Lun Lin and Tsung-Wei Huang. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2020.

[37] Dian-Lun Lin and Tsung-Wei Huang. Efficient gpu computation using task graph parallelism. In *Euro-Par*, pages 435–450, 2021.

[38] Dian-Lun Lin and Tsung-Wei Huang. Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 33(11):3041–3052, 2022.

[39] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, and Tsung-Wei Huang. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *ACM International Conference on Parallel Processing (ICPP)*, 2022.

[40] Y. Lin, W. Li, J. Gu, H. Ren, B. Khailany, and D. Z. Pan. ABCDPlace: Accelerated Batch-based Concurrent Detailed Placement on Multi-threaded CPUs and GPUs. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2020.

[41] S. Liu, P. Liao, Z. Chen, W. Lv, Y. Lin, and B. Yu. FastGR: Global Routing on CPU-GPU with Heterogeneous Task Graph Scheduler. *IEEE/ACM DATE*, 2022.

[42] Yi-Shan Lu and Keshav Pingali. Can Parallel Programming Revolutionize EDA Tools? *Advanced Logic Synthesis*, 2018.

[43] Sparsh Mittal and Jeffrey S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.*, 47(4), 2015.

[44] Gi-Joon Nam. Ispd 2006 placement contest: Benchmark suite and results. In *ACM ISPD*, 2006.

[45] Ulf Schlichtmann, Sabya Das, Ing-Chao Lin, and Mark Po-Hung Lin. Overview of 2019 cad contest at iccad. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 1–2, 2019.

[46] Shumpei Shiina and Kenjiro Taura. Almost Deterministic Work Stealing. In *ACM SC*, 2019.

[47] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting (Tim) Cheng. *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann Publishers Inc., 2009.

[48] Zhongjiang Yan. Conflict Graph Based Concurrent Transmission Scheduling Algorithms for the Next Generation WLAN. 25(5):1873–1885, 2020.

[49] Han Zhao, Quan Chen, Yuxian Qiu, Ming Wu, Yao Shen, Jingwen Leng, Chao Li, and Minyi Guo. Bandwidth and Locality Aware Task-Stealing for Manycore Architectures with Bandwidth-Asymmetric Memory. *ACM Trans. Archit. Code Optim.*, 15(4), 2018.