A Hybrid Scheduling Scheme for Parallel Loops

Aaron Handleman Arthur G. Rattew I-Ting Angelina Lee
Washington University in St. Louis
{ahandleman, arthur.rattew, angelee}@wustl.edu

Tao B. Schardl
Massachusetts Institute of Technology
neboat@mit.edu

***Abstract*—Parallel loops are commonly used parallel constructs to parallelize high-performance scientific applications. In the paradigm of task parallelism, the parallel loop construct is used to express the logical parallelism of the loop, indicating that the iterations in a loop are logically in parallel and let an underlying runtime scheduler determines how to best map the parallel iterations onto available processing cores.**

Researchers have investigated multiple scheduling schemes for scheduling parallel loops, with the static partitioning and dynamic partitioning being most prevalent. Static partitioning obtains low scheduling overhead while potentially retaining locality benefit in iterative applications that perform a sequence of parallel loops that access the same set of data repeatedly. But static partitioning may perform poorly relatively to dynamic partitioning if the loop iterations contain unbalanced workloads or if the cores can arrive at the loops in different times.

We propose a hybrid scheduling scheme, which first schedules loops using static partitioning but then employs dynamic partitioning when load balancing is necessary. Moreover, the work distribution employs a claiming heuristic that allows a core to check for partitions to work on in a semi-deterministic fashion, allowing the scheduling to better retain data locality in the case of iterative applications. Unlike prior work that optimizes for iterative applications, our scheme does not require programmer annotations and can provide provably efficient execution time. In this paper, we discuss the hybrid scheme, prove its correctness, and analyze its scheduling bound. We have also implemented the proposed scheme in a Cilk-based work-stealing platform and experimentally verified that the scheme load balances well and can retain locality for such iterative applications.

***Index Terms*—static partitioning, dyanmic partitioning, work stealing, work, span**

I. Introduction

Parallel loops are commonly used parallel constructs to parallelize high-performance scientific applications. Take the NAS benchmarks for example [1], most benchmarks are parallelized using only parallel loops as many applications are numeric in nature and have ample data parallelism.

In the paradigm of task parallelism, the parallel loop construct is used to express the logical parallelism of the loop, indicating that the iterations in a loop are logically in parallel and let an underlying runtime scheduler determines how to best map the parallel iterations onto available processing cores. In practice, three strategies are commonly implemented in various task-parallel platforms for shared-memory multicore machines: a) static partitioning [2], [3], b) dynamic fixed-size partitioning [2], [4]–[13], and c) guided partitioning [2]. Each of these strategies makes different trade-offs between loop allocations, load balancing, and synchronization overhead.

With ***static partitioning***, given a parallel loop with N iterations running on P processing cores, the N iterations are evenly divided into roughly equal-sized P partitions, with each partition assigned to a processing core. The partitioning is static because once a partition is assigned to a core, no other cores can take part in the work belonging to the partition. With ***dynamic partitioning***, on the other hand, the iteration space is divided into C fixed-size small chunks, where each ***chunk*** is a set of small number of iterations that execute sequentially on one core (i.e., typically $P \ll C$), and the runtime dynamically distributes the C chunks to the available P cores. Different platforms may utilize different scheduling schemes to distribute the work. Some platforms utilize ***work sharing***, where a centralized queue is used to hold the chucks, and cores synchronize on accesses to the queue in order to claim a chunk of work. An alternative is ***work stealing***, where each core maintains its own local queue of work available; when the need for load balancing arises, the cores "steals" from each other to move work from one queue to another, thereby balancing out the workload. Still, the chunk size, or the number of consecutive iterations that execute sequentially on one core, remains fixed. Finally, with ***guided partitioning***, the iteration space is divided into variable-sized chunks, where the chunk size starts out large and decreases until it reaches some small default size.

When executed on a multicore machine with shared memory, multiple factors can impact the performance of a parallel loop: allocation of loop iterations (where an iteration is executed), how well the underlying runtime load balances the loop, and synchronization / parallel overhead for enabling the distribution of the loop iterations. Each of these schemes make different trade-offs in these dimensions.

Static partitioning incurs small synchronization / parallel overhead for enabling the distribution of the loop iterations, since little scheduling decisions need to be made dynamically. Moreover, it tends to work well for applications where the loop iterations do in fact have an ***affinity*** for a particular core / processor, such as ***iterative applications*** where the application consists of multiple phases and each phase is encoded using a parallel loop. Assuming each phase accesses similar data, then iterations do have affinity — scheduling iterations that

This research was supported in part by National Science Foundation under grants XPS-1533644, CCF-1527692, CCF-1733873, CCF-1910568, and CCF-1943456.

Arthur G. Rattew conducted this research when he was an undergraduate student at Washington University in St. Louis; he is currently a Ph.D. student at the University of Oxford.

share an index across parallel loops naturally retains data locality. The static partitioning works well for such loops with affinity because given the same number of iteration count, the static scheme always distributes the loop iterations in a deterministic fashion, thereby keeping iterations with the same index on the same core across parallel loops. However, since the partitioning is static, if the workloads are unbalanced across the iterations, the core that finishes its partition last ends up dictating the overall execution time.

Dynamic partitioning, in comparison, incurs higher synchronization / parallel overhead in order to enable dynamic load balancing; thus, it provides much more robust performance when the iterations contain unbalanced workloads. Moreover, depending on the implementation, a task-parallel platform can schedule multiple parallel regions at the same time such that not all P are always available to execute a given parallel loop. In such a scenario, dynamic load balancing can be important even if the iterations contain the same workloads because different cores can arrive at the parallel loop at different time. Nevertheless, due to the nature of dynamic load balancing (where the result of the loop distribution depends on scheduling), dynamic partitioning does not account for loop affinity and cannot easily exploit the inherent data locality that exists in such iterative applications.

Guided partitioning allows for some dynamic load balancing, but in a different way than dynamic partitioning that makes different tradeoffs. In a scheduler that utilizes work sharing, the iteration space is divided into decreasing chunk sizes, and chunks are inserted a centralized queue. Cores synchronize accesses to the shared queue. Once a core obtains a chunk, it executes the chunk to completion before coming back to the queue. Thus, like dynamic partitioning, the allocation of iterations depends on scheduling and cannot easily exploit the inherent locality that exists in the iterative applications. However, due to the way iterations spaces are divided, it generates fewer chunks than dynamic partitioning, leading to slightly less synchronization / parallel overhead. Guided partitioning is not typically implemented in a work-stealing environment, where a distributed steal protocol is employed for load balancing, and therefore difficult to maintain centralized information necessary to determine chunk sizes.

In practice, different platforms implement different strategies. OpenMP offers different strategies that the programmer can choose by using the right compiler pragma [2]; it utilizes work sharing for its dynamic and guided partitioning. FastFlow [3], a C++-library based task-parallel platform, supports static and dynamic partitioning with work sharing [14]. Finally, many task-parallel platforms including TBB [5], variants of Cilk [6]–[9], variants of Habanero [10], [11], X10 [13] etc., employs dynamic partitioning with work stealing.

This paper proposes a ***hybrid partitioning*** scheme that combines static and dynamic partitioning with work stealing. At a high level, a hybrid loop is scheduled first using static partitioning, where the iteration space is divided into P partitions with one partition designated for each core. When a ***worker*** thread, a surrogate of a processing core, arrives at the execution of the loop (either because the worker starts the loop or happens to steal into it), the worker first tries to ***claim*** a partition designated to it based on the static partitioning. To accommodate unbalanced iterations and the fact that not all workers may arrive at the loop at the same time, once the worker finishes its designated partition, it utilizes a ***claiming heuristic*** that checks for the next partition to work on in a semi-deterministic fashion. Using this claiming heuristic, a worker will check whether it can claim the other partitions in a *deterministic* sequence, computed based on its worker ID. The heuristic produces partitions to work on as long as the checked partition has not been claimed by any other worker. After performing the claiming heuristic, a worker then reverts back to dynamic partitioning using classic work stealing.

Some nondeterminism in how the iterations are scheduled can still play out but it helps with retaining loop affinity and at the same time allows for dynamic load balancing. Moreover, we show that the proposed claiming heuristic ensures that each partition executes exactly once and incurs at most $\lg P$ failed claims per worker before a worker moves onto dynamic partitioning using classic work stealing. Given a parallel loop with N iterations, a work-stealing scheduler with dynamic partitioning can schedule the loop in expected time $T_1/P + O(\lg N + max_{i=0}^{N}(T_\infty(i))$, where T_1 is the time it takes to execute the loop on one core, and $max_{i=0}^{N}(T_\infty(i))$ denotes the maximum span of any iteration of the loop. In contrast, the hybrid scheme executes in expected time $T_1/P+O(P+\lg N+max_{i=0}^{N}(T_\infty(i))$, which helps with loop affinity with an additive $O(P)$ overhead.

Researchers have studied scheduling mechanisms to trade off between loop affinity and load balancing. Some approaches adjust chunk sizes [15]–[24] which requires some centralized control (e.g., requires a centralized queue or needs to know which processor is most loaded). We took a different approach by using a distributed work-stealing protocol that allows partitioning to adjust dynamically without maintaining centralized information. Others have investigated improving locality in iterative applications running on a work-stealing platform [25], [26]. These works require programmer annotations and do not focus on providing a provable execution time bound.

We have implemented the hybrid partitioning scheme in OpenCilk [9], an open-source version of Cilk that comes with a compiler and a work-stealing runtime scheduler. We experimentally evaluate the proposed scheme by comparing it against OpenMP, FastFlow, and Cilk with dynamic partitioning. Our results indicate that the proposed scheme performs comparably to static partitioning when the iterations have balanced workload and incurs lower overhead and thus better scalability compared to other partitioning schemes when the iterations have unbalanced workload.

In summary, this paper makes the following contributions:

- We propose a hybrid partitioning scheme that combine static and dynamic partitioning with a semi-deterministic claiming heuristics (Section III).
- We show that the claiming heuristic is correct (every partition executes exactly once) and that the hybrid scheme

executes a loop in expected time $T_1/P + O(P + \lg N + max_{i=0}^{N}(T_\infty(i))$ (Section IV).
- We empirically evaluate the hybrid scheme and show that it performs comparably or better depending on the workload compared to other schemes (Section V).

II. Preliminaries

This section briefly summarizes the necessary background. Since our hybrid scheme is implemented by extending a variant of Cilk [9], which utilizes a work-stealing scheduler, we briefly summarize the task-parallel constructs supported in Cilk, how its scheduler operates, and how we analyze the performance of a Cilk computation.

The basic Cilk supports three keywords to allow for parallel execution: `spawn`, `sync`, and `cilk_for`. When a function F ***spawns*** another function G (preceding the invocation with the `spawn` keyword), it indicates that the continuation of F after the `spawn` statement can potentially execute in parallel with the invocation of G. The `sync` statement is the counterpart, which indicates that all previously spawned child subroutines must return before the control can pass `sync`. The `cilk_for` keyword indicates that iterations of the loop are logically in parallel. In Cilk, a loop parallelized with `cilk_for` is implemented using the divide-and-conquer strategy that performs binary spawning on the iteration space until the number of iterations reaches a particular chunk size, in which case a sequential version (i.e., containing no parallelism) of the loop is invoked to process those iterations. By default, the chunk size is set to the minimum of 2048 and $N/8P$ where N is the number of iterations and P the number of cores used.

Cilk schedules a parallel computation using a work-stealing scheduler, which works as follows. At the runtime startup, P workers are created, where each worker is a surrogate of a processing core. Each worker has its own deque holding available work, and for the most part a worker operates only on its own deque. When a worker executes F that spawns G, the frame for F is pushed onto the bottom of its deque. When a worker returns from G, it pops the parent frame F off its deque and continue execution of F, assuming that F has not been stolen. The one-core execution of a Cilk computation mirrors that of its ***serial counterpart*** (the sequential code obtained by removing `spawn`, `sync`, and converting `cilk_for` to `for`). The behavior of a worker diverges, however, when it runs out of work — it turns into a ***thief*** and randomly chooses another ***victim*** worker to steal from. A thief always steals the top-most frame from the victim, and a successful resulted in actual parallelism, where the continuation of the function corresponding to the stolen frame now executes in parallel (by the thief) with its subroutine (by the victim).

A parallel execution can be modeled as a ***computation dag*** [27], where each node denotes a sequence of instructions containing no parallel control and each edge denotes a dependence — a node cannot execute until all its predecessors have executed. When F spawns G, the `spawn` terminates the current node and forms two outgoing edges, one to the instance of G spawned and one to the continuation of F. When F invokes a `sync`, it forms a node with multiple incoming edges, one from each of the spawned subroutines and one from the last continuation, where the node denotes the continuation after the `sync` statement.

A work stealing scheduler provides provably good execution time bound [27]. To state the bound, there are two important performance metrics that one cares about. First is the ***work***, denoted as T_1, or the time it takes to execute the all the nodes in the dag on one core. Second is the ***span***, denoted as T_∞, or the time it takes to execute a longest chain of dependences in the computation dag. Given a computation with work T_1 and T_∞, Cilk's work-stealing scheduler can execute the computation in expected time $T_1/P + O(T_\infty)$ when running on P cores.

To state the execution time bound of a `cilk_for` loop, we have to consider how it is implemented. A `cilk_for` is effectively translated into a recursive function that spawns off itself with the first half of the iteration space and calls the itself in the continuation to process the second half, until the function reaches the base case where the number of iterations reaches the chunk size. This divide-and-conquer recursive binary spawning incurs a little overhead. Specifically, a `cilk_for` with N iterations and a constant chunk size has the span of $\lg N + max_{i=0}^{N}(T_\infty(i))$, where $max_{i=0}^{N}(T_\infty(i))$ denotes the maximum span of any iteration of the loop.

III. The hybrid loop Scheduling Scheme

This section overviews the scheduling scheme for the proposed hybrid loop. To simplify the presentation of the hybrid loop, we shall first assume that the number of workers P is always a power of 2. Then we discuss how to generalize hybrid loops to avoid this assumption.

Overview of the implementation

A hybrid loop augments standard randomized work stealing with a custom algorithm for mapping iterations to workers. Conceptually, work in a hybrid loop is divided into P equally sized partitions, with each partition earmarked for a particular worker. To handle load imbalance or different worker (loop) starting time, the hybrid loop algorithm employs a claiming heuristic for dynamic load balancing in a way that accounts for loop affinity.

Algorithm 1: Pseudocode for `InitHybridLoop`

```
Data: Start index start, end index (exclusive) end, number
      of partitions R
1 A ← initPartition(start, end);
2 spawn DoHybridLoop(my_wid(), R, A);
3 sync;
```

To describe the implementation of a hybrid loop, we first discuss how a hybrid loop is compiled. Algorithms 1, 2, and 3 present the pseudocode for the compiler-runtime ABI of a hybrid loop's implementation. Conceptually, a hybrid loop is compiled into the code shown in Algorithm 1. This code first

Algorithm 2: Pseudocode for `Claim`

```
   Data: Index i, worker ID w, partition data structure A
4  r ← i ⊕ w;
5  if fetch_and_or(A[r], 1) then
6  |  return 0;
7  else
8  |  return 1;
```

Algorithm 3: Pseudocode for `DoHybridLoop`

```
   Data: Worker ID w, number of partitions R, partition data
         structure A
9  i ← 0;
10 if Claim(i, w, A) then
11 |  spawn doWork(i ⊕ w);
12 |  i ← i + 1;
13 else
14 |  return;
15 while i < R do
16 |  if Claim(i, w, A) then
17 |  |  spawn doWork(i ⊕ w);
18 |  |  i ← i + 1;
19 |  else
20 |  |  i ← i + (i & −i); /* increment i by     */
   |  |  /* its least-significant set bit   */
21 sync;
22 return;
```

initializes a partition data structure A necessary to execute the hybrid loop heuristic (line 1), where the iteration space is evenly divided into P partitions, each designated to a worker, where P is the number of workers. The code then spawns off a function named `DoHybridLoop` (line 2) implemented by the runtime system.

Algorithms 2 and 3 present pseudocode for `DoHybridLoop`, and an auxiliary method `Claim`, that implement the claiming heuristic used by a hybrid loop to account for loop affinity. The heuristic partitions the iterations of a loop into $R = 2^k$ partitions, for some integer $k \geq 0$. The heuristic uses a worker-specific mapping of each partition to an ***index number*** $i \in \{0, 1, 2, \ldots, R-1\}$, using that worker's ID $w \in \{0, 1, 2, \ldots, P-1\}$. For simplicity, we shall refer to a worker interchangeably with that worker's ID. For worker w, line 4 in Algorithm 2 maps each index i to a partition $r = i \oplus w$, where $\oplus$ is the bitwise XOR operator. Because $\oplus$ is bijective and $\oplus$ is its own inverse, for each worker w, each partition r maps to a distinct index $i = w \oplus r$, and each index i maps to a distinct partition $r = w \oplus i$. We say that a worker w ***successfully*** claims a partition r if a call to `Claim(i, w, A)`, where $i = r \oplus w$, returns 1 (line 8). Otherwise the claim is ***unsuccessful*** when it returns 0 (line 6).

Since each worker w entering the heuristic in Algorithm 3 starts with $i = 0$ and its own unique worker ID w, a worker w effectively tries to first claim the partition earmarked for it. If this claim fails, some other worker w' has claimed it and w moves onto dynamic load balancing with work stealing. On the other hand, if this claim succeeds, w executes the iterations in the claimed partition via `doWork`, updates i (line 12), and continues on with the heuristic (lines 15–20). As the heuristic continues, w tries to claim other partitions by updating i (line 18 or line 20), which generates a claim sequence unique to worker w.

The body of the original loop is lifted into the routine `doWork`, which is called on lines 11 and 17 to execute a partition of iterations. In particular, this routine performs an ordinary divide-and-conquer parallel loop with binary spawning as described in Section II over the iterations of a partition, to allow for dynamic load balancing of the work of a partition.

We now address the simplifying assumption that the number of workers P is a power of 2. If P is not a power of 2, the hybrid loop uses R equal to the next power of 2 greater than P, and only earmarks the first P partitions for the workers. The `DoHybridLoop` routine ensures that these additional unassociated partitions are nevertheless executed.

Steal protocol for `DoHybridLoop` frames

Recall from Section II that each worker maintains its own deque of frames and utilizes steals to load balance. The randomized work-stealing scheduler handles a `DoHybridLoop` frame differently from other frames in the runtime. First, a `DoHybridLoop` frame contains a pointer to a partition data structure A, which keeps track of the process of the loop. Second, a worker stealing a `DoHybridLoop` frame follows a custom protocol for stealing work.

When a worker w attempts to steal a `DoHybridLoop` frame, it first checks whether its designated starting partition, $r = w \oplus 0$, has been claimed, that is, if $A[r]$ is 1. If the partition has been claimed, then the worker performs ordinary randomized work stealing. If the partition has not been claimed, then the worker creates its own copy of a `DoHybridLoop` frame executing `DoHybridLoop` (w, R, A), that is, a `DoHybridLoop` frame executing with the same arguments R and A but with the worker ID w of the thief. As a result, if the worker's designated starting partition r is available, then the worker starts executing `DoHybridLoop` in a manner so as to start by executing partition r.

This `DoHybridLoop` steal protocol means that, when stealing a `DoHybridLoop` frame, a worker may begin executing a different partition than the partition that the victim would normally execute next. As a result, a partition may execute out-of-order, compared to the order implied by the fork-join code in Algorithms 1, 2, and 3 executed by the worker that started the hybrid loop. For example, suppose that worker 0 starts executing a hybrid loop. If no workers steal from worker 0, then worker 0's call to `DoHybridLoop` spawns off the execution of the partitions in order 0, 1, 2, etc. But if worker 2 steals the `DoHybridLoop` frame from worker 0 before worker 0 claims partition 1, then worker 2 starts executing partition 2, before worker 0 spawns execution of partition 1. Section IV proves the correctness of this protocol, as well as its worst-case parallel performance.

IV. The Analysis of the Hybrid Loop Scheduler

This section analyzes the hybrid loop algorithm described in Section III. We prove its correctness of hybrid loop and

analyze its running time a hybrid loop using work/span analysis [28, Ch. 27]. This analysis shows that, when the number of partitions matches the number of workers P, then the worst-case parallel running time of a hybrid loop on n iterations is $T_P \leq T_1/P + O(P + \lg n + max_{i=0}^{n}(T_\infty(i))$, where T_1 is the total work of the hybrid loop and $max_{i=0}^{n} T_\infty(i)$ denotes the maximum span of any iteration of the loop.

Correctness

To justify the correctness of hybrid loop, we first argue that the `Claim` routine in Algorithm 2 ensures that any partition that any worker attempts to claim is executed exactly once. We then argue that the execution of `DoHybridLoop` by any nonempty set of workers ensures that each partition is claimed. As a result, we conclude that the hybrid loop heuristic ensures that every partition is executed exactly once.

The following lemma shows that, after some worker w calls `Claim(`*i, w, A*`)` for index i, the partition $i \oplus w$ is guaranteed to be executed to be executed exactly once.

Lemma 1. *After a worker w attempts to claim a partition $r = w \oplus i$ for some index i, partition r has been executed exactly once.*

Proof. Consider the pseudocode in Algorithms 2 and 3. If worker w executes the `fetch_and_or` instruction on line 5 and it succeeds, then w executes partition r on on line 11 or line 17. Otherwise the flag $A[r]$ for partition r was previously set to 1, which means another worker w' must have successfully executed line 5, in which case w' executes r. □

To argue the correctness of the hybrid loop, for each worker w, we consider the set of R partitions in a hierarchy of groups, based on the index that maps to each partition. For $0 \leq n \leq k$, define the ***level-n index group*** to be the set $I(x,n) = \{x \cdot 2^n, (x \cdot 2^n) + 1, (x \cdot 2^n) + 2, \ldots, (x \cdot 2^n) + 2^n - 1\}$ of 2^n of indices, where $x \in \{0, 1, 2, \ldots, R/2^n - 1\}$. For example, suppose we have $R = 2^3 = 8$ indices, $\{0, 1, 2, 3, 4, 5, 6, 7\}$. These indices subdivide into the level-1 index groups $\{0, 1\}$, $\{2, 3\}$, $\{4, 5\}$, and $\{6, 7\}$, and they also subdivide into the level-2 index groups $\{0, 1, 2, 3\}$ and $\{4, 5, 6, 7\}$. Index groups have the following properties:

- For $n > 0$, we have $I(x,n) = I(2x, n-1) \cup I(2x+1, n-1)$.
- For $n < 2^k$, we have that $I(x,n)$ is a subset of a single level-$n+1$ index group, $I(\lfloor x/2 \rfloor, n+1)$.

For each level-n index group and worker w, define the ***level-n partition group*** to be the set $G(w,x,n) = \{w \oplus (x \cdot 2^n), w \oplus ((x \cdot 2^n) + 1), \ldots, w \oplus ((x \cdot 2^n) + 2^n - 1)\}$, that is, the set obtained by computing $w \oplus i$ for every $i \in I(x,n)$. For convenience, we denote the partition group $G(w,x,n) = w \oplus I(x,n)$. For instance, for worker $w = 5$ and the previous example of $R = 8$ indices, the level-2 partition groups are $5 \oplus \{0, 1, 2, 3\} = \{5, 4, 7, 6\}$ and $5 \oplus \{4, 5, 6, 7\} = \{1, 0, 3, 2\}$. In addition, we say a worker w attempts to claim index group $I(x,n)$ when it attempts to claim partition $w \oplus (x \cdot 2^n)$, that is, when w attempts to claim the partition mapped to the first index in $I(x,n)$.

The following lemma analyzes the pseudocode in Algorithm 3 to justify that all partitions are claimed.

Lemma 2. *If any worker attempts to claim a partition within a partition group, then all partitions in that group are claimed.*

Proof. The proof follows by induction on the level n of the partition group. Consider the pseudocode in Algorithm 3, and suppose that a worker w is claiming a partition in a partition group $G(w,x,n)$ for some x.

In the base case, $n = 0$ and partition group $G(w,x,0)$ contains exactly one partition. When that partition is claimed on line 10 or line 16, all partitions in $G(w,x,0)$ are claimed.

For $n > 0$, we suppose the lemma holds for level $n-1$. Lines 12 and 18 implies that a worker claims an index group $I(x,n)$ by first recursively claiming index $I(2x, n-1)$ and then recursively claiming index $I(2x+1, n-1)$. We consider the possible outcomes.

Suppose first that worker w succeeds in recursively claiming $I(2x, n-1)$. By the inductive hypothesis, all partitions in $G(w, 2x, n-1) = w \oplus I(2x, n-1)$ are claimed when partition $r = w \oplus (2x \cdot 2^{n-1})$ is claimed. When w subsequently attempts to recursively claim $I(2x+1, n-1)$, then whether or not w succeeds, by the inductive hypothesis, every partition in $w \oplus I(2x+1, n-1)$ is claimed.

Suppose instead that w fails to recursively claim $I(2x, n-1)$. Line 20 increments the current index by its least-significant set bit. This code implies that w next attempts to recursively claim an index group at a higher level, which does not contain index group $I(2x+1, n-1)$. We shall see that all partitions in $G(w,x,n) = w \oplus I(x,n)$ are nevertheless claimed.

Let worker w' be the worker that successfully claimed partition $y = w \oplus (2x \cdot 2^{n-1})$, thus causing worker w to fail to claim $I(2x, n-1)$. Partition y is contained in some level-$n-1$ partition group $G(w', x', n-1) = w' \oplus I(x', n-1)$ for some $x' \in \{0, 1, 2, \ldots, R/2^{n-1} - 1\}$.

We first show that all partitions in $G(w, 2x, n-1)$ are also in $G(w', x', n-1)$. Consider a partition $r \in G(w, 2x, n-1)$, and let $v = w' \oplus w$. For some $a \in \{0, 1, 2, \ldots, 2^{n-1} - 1\}$, we observe that

$$\begin{aligned} r &= w \oplus ((2x \cdot 2^{n-1}) + a) \\ &= w' \oplus w' \oplus w \oplus ((2x \cdot 2^{n-1}) + a) \\ &= w' \oplus ((\lfloor v/2^{n-1} \rfloor \oplus 2x) \cdot 2^{n-1} + (v \bmod 2^{n-1} \oplus a)) \\ &= w' \oplus ((x' \cdot 2^{n-1}) + a') \ , \end{aligned}$$

where $x' = \lfloor v/2^{n-1} \rfloor \oplus 2x$ and $a' = v \bmod 2^{n-1} \oplus a$. Because $a' \in \{0, 1, 2, \ldots, 2^{n-1} - 1\}$, we have $r \in w' \oplus I(x', n-1)$, and thus $r \in G(w', x', n-1)$. Hence, $G(w, 2x, n-1) = G(w', x', n-1)$. Because the inductive hypothesis implies that all partitions in $G(w', x', n-1)$ are claimed, all partitions in $G(w, 2x, n-1)$ are claimed.

To argue that the partitions in $G(w, 2x+1, n-1)$ are also claimed, we consider two cases, based on the parity of x'.

Case 1: x' is even. Because $x' = \lfloor v/2^{n-1} \rfloor \oplus 2x$, we have $x'+1 = \lfloor v/2^{n-1} \rfloor \oplus (2x+1)$, which implies that any partition in $G(w, 2x+1, n-1)$ is also in $G(w', x'+1, n-1)$. Because worker w' attempts to recursively claim $G(w', x'+1, n-1)$ after $G(w', x', n-1)$, by the inductive hypothesis, all partitions in $G(w, 2x+1, n-1)$ are claimed.

Case 2: x' is odd. Because $x' = \lfloor v/2^{n-1} \rfloor \oplus 2x$ and $2x$ is even, we have that $\lfloor v/2^{n-1} \rfloor$ is odd. Therefore, $\lfloor v/2^{n-1} \rfloor \oplus (2x+1) = x' - 1$, and any partition in $G(w, 2x+1, n-1)$ is also in $G(w', x'-1, n-1)$. Because worker w' claims the partitions in group $G(w', (x'-1)/2, n)$ by recursively claiming $G(w', x'-1, n-1)$ followed by $G(w', x', n-1)$, worker w' must have already recursively claimed $G(w', x'-1, n-1)$. Therefore, by the inductive hypothesis, all partitions in $G(w, 2x+1, n-1)$ are claimed. □

Lemmas 1 and 2 combine to complete the correctness proof.

Theorem 3. *Every partition in a partition group is executed exactly once.*

Proof. Lemma 2 implies that every partition is claimed, and Lemma 1 implies that each claimed partition is executed exactly once. □

Running-time analysis

We now analyze the parallel running time of a hybrid loop, based on the fork-join dependencies of Algorithms 1, 2, and 3. Because the steal protocol for a `DoHybridLoop` implies that a partition may be executed sooner than implied by these fork-join dependencies, this analysis provides a worst-case bound on the parallel running time of a hybrid loop.

To begin, the following lemma bounds the work involved in successfully claiming a partition.

Lemma 4. *After at most* $\lg R$ *unsuccessful claims, a worker either performs a successful claim or returns from* `DoHybridLoop`*.*

Proof. As Algorithm 3 shows, when a call to `Claim` returns 0, either line 14 or line 20 executes. Line 14 causes the worker to exit `DoHybridLoop` immediately, whereas line 20 increments the current index i by the least significant set bit in the binary representation of i. After at most $\lg R$ executions of line 20 without a successful call to `Claim` on line 16, i must be at least R. In this case, the worker exits the loop on lines 15–20 and return on line 22. □

We now consider the steal protocol on `DoHybridLoop` frames, which allows a worker upon stealing a `DoHybridLoop` frame to begin executing a partition other than the partition that the continuation of the `DoHybridLoop` frame would execute. We analyze the effect of this `DoHybridLoop` steal protocol.

We first note that the `DoHybridLoop` steal protocol executes in $\Theta(1)$ work. When a worker w begins executing `DoHybridLoop`— either by executing line 2 or by stealing a `DoHybridLoop` frame — it checks the A structure in $\Theta(1)$ work to check if its designated partition is available. If so, w creates its own copy of the `DoHybridLoop` frame onto its deque, in $\Theta(1)$ work, in which it invokes `DoHybridLoop` using its own worker ID. Otherwise, the worker performs ordinary randomized work stealing.

To study how this steal protocol affects the theoretical performance of the scheduler, we model the execution of a hybrid loop using a nondeterministic computation dag as follows. When a worker steals a continuation of a `DoHybridLoop` frame — which corresponds with the continuation of lines 11 and 17 — and begins executing partition r, we model the execution of r as a successor of the corresponding `spawn` node in the computation dag. Thus, unlike an ordinary `spawn` node, this node can end up with more than two successors. Because the `DoHybridLoop` steal protocol allows at most P steals from `DoHybridLoop` frames for a hybrid loop to behave differently from normal steals, the `DoHybridLoop` steal protocol nondeterministically produces a tree of $O(P)$ nodes in the computation dag to model how the partitions of a hybrid loop are enabled for parallel execution.

Although the subdag modeling a hybrid loop is nondeterministic, the worst case — in which the subdag has the longest span — occurs when the processing of each partition of the hybrid loop is spawned off serially, as described by the fork-join code in Algorithms 1, 2, and 3. Therefore, to bound the parallel running time of a hybrid loop, the following theorem analyzes the hybrid loop according to the fork-join dependencies implied by this fork-join code and ordinary randomized work stealing.

Theorem 5. *Consider a loop of* n *iterations divided into* $R < n$ *partitions and scheduled using a hybrid loop. For* $j \in \{0, 1, 2, \ldots, n-1\}$*, let* $T_1(j)$ *and* $T_\infty(j)$ *denote the work and span, respectively, of the* j*th iteration of the loop. Then the parallel loop executes on* P *workers in time* $T_P \leq \sum_{j=0}^{n} T_1(j)/P + \Theta(R + n/R)/P + O(R \lg R)/P + O(R + \lg n + \max_{j=0}^{n} \{T_\infty(j)\})$.

Proof. We analyze the execution of a hybrid loop using work/span analysis [28, Ch. 27].

Initializing a partition data structure A, on line 1 of Algorithm 1, takes $\Theta(R)$ work and $\Theta(\lg R)$ span.

Each call to `Claim` performs constant work and is either successful or unsuccessful. Lemma 4 shows that any worker performs at most $\lg R$ unsuccessful claims before successfully claiming a partition or returning to ordinary randomized work stealing. Hence a total of $O(R \lg R)$ work is incurred to claim all R partitions, and $O(\lg R)$ span is incurred to claim a single partition.

The iterations in each partition are executed using a `cilk_for` loop, to support efficient dynamic load balancing of partitions with different amounts of work. Therefore, after some worker w successfully claims a partition r, the execution of r incurs work $\Theta(n/R) + \sum_{j \in r} T_1(j)$ and span $\Theta(\lg(n/R)) + \max_{j \in r} T_\infty(j)$. Hence, the loop on lines 15–20, which, in the worst case, iteratively spawns the execution of each partition, incurs work $\Theta(R + n/R) + \sum_{j \in r} T_1(j)$ and span $O(R) + \Theta(\lg(n/R)) + \max_{j \in r} T_\infty(j)$.

Combining these analyses, executing a hybrid loop incurs work $\sum_{j=0}^{n} T_1(j) + O(R \lg R) + \Theta(R + n/R)$ and span $\max_{j=0}^{n} T_\infty(j) + O(R) + \Theta(\lg n)$. Plugging these terms into the parallel running-time bound T_P yields the bound. □

The following corollary, which follows from Theorem 5, analyzes the common case where $R = P$.

Corollary 6. *When the number of partitions R is set equal to the number of workers P and $n > P$, a hybrid loop with work $T_1(j)$ and span $T_\infty(j)$ per iteration j runs on P workers in time $T_P \leq \Theta(\sum_{j=0}^{n} T_1(j))/P + O(P + \lg n + \max_{j=0}^{n} \{T_\infty(j)\})$.*

V. Empirical Evaluation

In this section, we empirically evaluate our proposed scheduling scheme for hybrid loops against static partitioning, dynamic partitioning, and guided partitioning implemented in various popular task-parallel platforms. We have evaluated these schemes using microbenchmarks and the five kernels from NAS parallel benchmarks [1] — since the original NAS benchmarks are based in Fortran (even the OpenMP port), we used the C/C++ port by Griebler et al. [29] which included the five kernels based on NBP3.3.1 with ports to OpenMP and FastFlow, two platforms that we compare to.[1] Empirical results indicate that our proposed hybrid scheme performs comparably to static partitioning when the iterations have balanced workload and incurs lower overhead and thus better scalability compared to other partitioning schemes when the iterations have unbalanced workload. Moreover, our hybrid scheme tends of retain loop affinity as well as the static partitioning strategy and retain loop affinity much better compared to other dynamic schemes.

Experimental setup

We ran our experiments on a 32-core machine with 2.20-GHz cores on four sockets (Intel Xeon E5-4620) (8-core per socket). Each core has a 32-KByte L1 data cache, 32-KByte L1 instruction cache, and a 256-KByte L2 cache. Each socket shares a 16-MByte L3-cache, and the overall size of DRAM is 512 GByte.

We compare to various loop scheduling schemes used in OpenMP [2] (`omp_static`, `omp_dynamic`, and `omp_guided` for OpenMP with static, dynamic, and guided partitioning respectively), FastFlow [3] (`ff`) and vanilla Cilk from the OpenCilk release [9] (`vanilla` with dynamic partitioning using work stealing). FastFlow supports both static and dynamic partitioning with work sharing. As its performance tends to lag behind other platforms, we ran FastFlow with both schemes and only displayed the data point with better performing scheme (always shown as (`ff`). For all platforms, we pin threads to cores in a compact fashion during executions, i.e., if less than 8 threads are used, only one socket is employed.

[1]The codebase that we adopted can be found at https://github.com/dalvangriebler/NPB-CPP.

All Cilk-based benchmarks are compiled with Tapir [30], a LLVM/Clang (version 7) based Cilk compiler. Others are compiled using the same version of LLVM/Clang that Tapir is based on. We used OpenMP 4.5 that came with LLVM/Clang. We obtained the FastFlow runtime via codebase found at https://github.com/fastflow/fastflow. We have used the vanilla Cilk Plus originally released by Intel [4] that came with the Tapir compiler. All software is compiled using `-O3` and run on Linux kernel version 4.15 with NUMA support enabled and dynamic frequency scaling and hyperthreading disabled. For all platforms, we have used NUMA-aware memory allocation to distribute the data across sockets to allow the static partitioning to exploit the locality benefit.

Benchmarks

We have constructed two microbenchmarks simulating iterative applications with heavy data accesses, one with balanced parallel loop iterations (`balanced`) and one with unbalanced (`unbalanced`). Each microbenchmark consists of an outer sequential loop with an inner parallel loop, where each parallel loop iteration operates on an array in strides of 13 modulo the size of the array (which prevents the prefetcher from prefetching on the machine we used). Each parallel iteration in the `balanced` accesses the same amount of data, whereas that parallel iterations in `unbalanced` access variable amounts. The arrays accessed by different parallel iterations do not overlap in memory.

We have used the five kernels from the NAS parallel benchmark suite [1], which can be mostly parallelized using parallel loops. The kernels include `mg`, a V-cycle multigrid algorithm, `ft`, fast Fourier transforms, `ep`, an embarrassingly parallel kernel that generates pairs of Gaussian random deviates, `is`, a sorting algorithm, and `cg`, conjugate gradient.

Results on microbenchmarks

Absolute performance. We ran the two microbenchmarks on three different working set sizes, one with which data accessed by cores on a socket is well within the L3 cache capacity ($<$ 12MB), one at about the L3 cache capacity ($<$ 16MB), and one that exceeds the L3 cache capacity but fits within the DRAM of a socket (266MB).

The plots in Figure 1 show the scalability results of the microbenchmarks, namely T_1/T_P, where T_1 is the running time on one core, including overhead necessary to enable parallelism, and T_P is the running time on P cores. The fist column shows the work efficiency, namely T_s/T_1, where T_s is the running time of the sequential code without any parallel constructs. Each data point corresponding to the running times used to compute the scalability plots are the average of 5 runs with standard deviation less than 4% except for a couple data points with standard deviation less than 5% (`hybrid` and `vanilla` for `unbalanced` running on 32 cores with the largest working set size).

The reason why we separately show T_s/T_1 and scalability (as opposed to speedup, T_P/T_s) is that it conveys more information. For configuration where the partitioning is done at

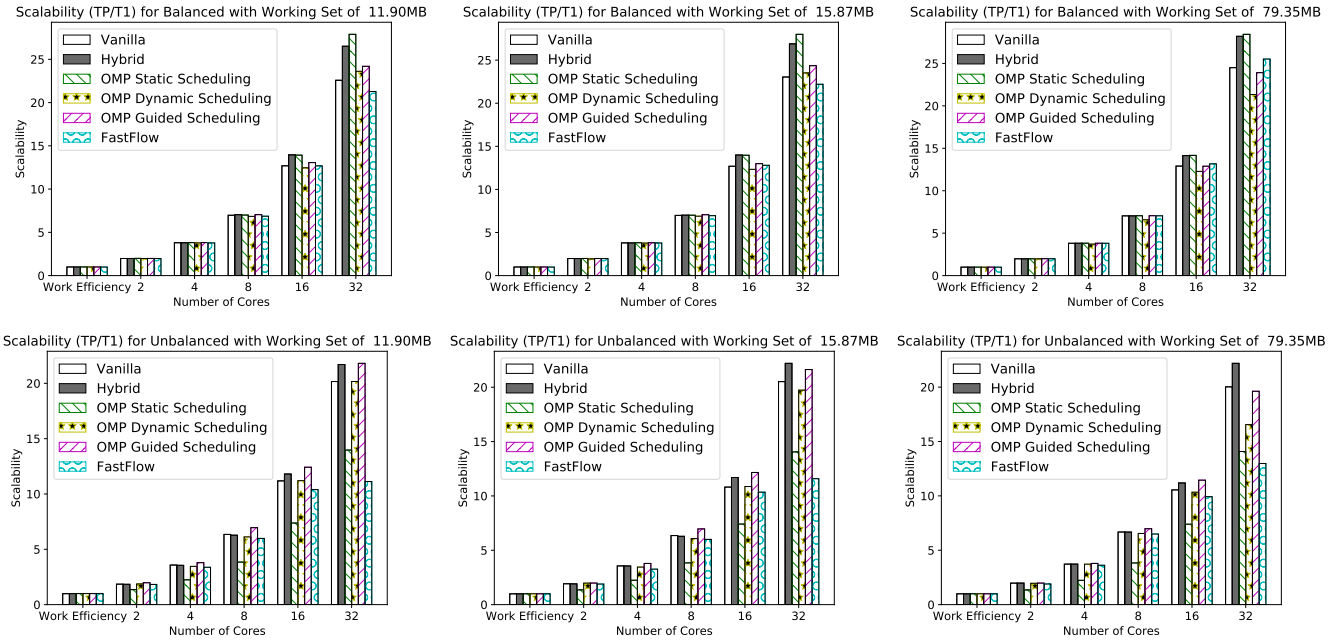

Fig. 1. The work efficiency and scalability results of running `balanced` (top) and `unbalanced` (bottom) microbenchmarks on three different working set sizes (from left to right, well under L3 cache size, at about L3 cache size, and above L3 cache size).

runtime (e.g., dynamic partitioning), the chunk size — number of consecutive iterations that can only be run on one core — can impact the work efficiency. By default most platforms (OpenMP and FastFlow) utilize chunk size of one for dynamic partitioning, which can incur high parallel overhead and causes the performance to suffer. We thus configured the platforms to use a larger chunk size (but small enough to still have sufficient parallelism). All platforms are run with the same chunk size of either $N/8P$ or 2048, whichever is smaller. As the work efficiency column indicates, all platforms are adjusted to have similar work efficiency (close to one, which is the desired value).

For the `balanced` workload, all platforms perform comparably when the P used are all within a single socket. Here, FastFlow (`ff`) shown uses the static partitioning scheme as it performs better than the dynamic scheme on the `balanced` workload. The performance differences become more evident when the executions are cross-sockets, i.e., on 16 and 32 cores. The `omp_static` configuration outperforms all the other platforms, which is not surprising, as the workload is designed to be well suited for static partitioning. The partitioning for `omp_static` is done primarily by the compiler and incurs little scheduling overhead (which mainly includes the runtime startup cost, which is amortized across the sequence of parallel loops as it is only done for the very first parallel loop). Nevertheless, the performance of our hybrid scheme follows closely. Our hybrid scheme incurs slightly more scheduling overhead, as the partitioning is done at runtime with some bookkeeping to allow for dynamic load balancing when necessary.

The performance for the non-static partitioning solutions (`vanilla`, `omp_dynamic`, and `omp_guided`) tend to lag behind when the executions cross sockets. The `omp_guided` tends to outperform `omp_dynamic` as both use work sharing and `omp_guided` incurs lower scheduling overhead (as threads access the shared queue less frequently). Cilk Plus (`vanilla`) uses work-stealing and lags behind for the smaller working sets but come slightly ahead in the larger working set. It is a little surprising that the performance of `ff` also lags behind in the smaller working set size, despite the fact that it uses static partitioning.

For the `unbalanced` workload, the FastFlow shown (`ff`) uses the dynamic partitioning with the adjusted chunk size. Here the non-static schemes (except for `ff`) clearly win out. It's a bit surprising how well `omp_guided` and `omp_dynamic` perform, especially for the smaller working set size. In general, the OpenMP implementations seem to have stronger performance when the loops are small (i.e., smaller working sets translate to less work in our microbenchmarks). As the working set size increases (and work increases), however, our hybrid scheme performs comparably or better. Similar trend can be observed between the `omp_dynamic` (dynamic partitioning with work sharing) and `vanilla` (dynamic partitioning with work stealing), where the work stealing variant performs better as the amount of work increases.

Retaining loop affinity. Finally, we use the microbenchmarks to evaluate how well our scheme retains loop affinity compared to other schemes. We focus on comparing the hybrid scheme against the dynamic partitioning in Cilk (`vanilla`) and the various OpenMP schemes (`omp_*`) and omit FastFlow, as the schemes used by FastFlow are also supported by OpenMP, and the OpenMP schemes performed better than the FastFlow ones on these microbenchmarks.

	11.90MB	15.87MB	79.35MB
hybrid balanced	99.99%	99.99%	99.99%
vanilla balanced	3.16%	3.15%	3.16%
omp_static balanced	100.00%	100.00%	100.00%
omp_dynamic balanced	9.31%	10.29%	11.95%
omp_guided balanced	4.83%	4.75%	4.64%
hybrid unbalanced	67.52%	67.31%	67.15%
vanilla unbalanced	3.19%	3.19%	3.19%
omp_static unbalanced	100.00%	100.00%	100.00%
omp_dynamic unbalanced	4.20%	4.21%	4.27%
omp_guided unbalanced	4.31%	4.27%	4.14%

Fig. 2. The percentage of loop iterations executed by the same core in consecutive parallel loops running microbenchmarks on 32 cores using our scheme (`hybrid`), the dynamic partitioning in Cilk (`vanilla`), and various OpenMP schemes (`omp_`$\star$).

Figure 2 shows the percentage of iterations that got executed by the same core in consecutive parallel loops (i.e., the fraction of the iterations such that an iteration j from loop i executed on the same core as the iteration j from loop $i-1$) when running on 32 cores. The percentage shown is the median out of three runs. As expected, the `omp_static` is able to maintain 100% loop affinity across parallel loops. Our `hybrid` scheme works well for the `balanced` workload and was able to retain 67% of loop affinity for the `unbalanced` workload. As such, our scheme was able to outperform `omp_static` for the `unbalanced` workload (per Figure 1) as it load balances better and can retain some locality at the same time. All the other dynamic schemes (`vanilla`, `omp_dynamic` and `omp_guided` had a hard time retaining loop affinity. As such, they perform obviously worse than `omp_static` for the `balanced` workload, but still win out for the `unbalanced` workload due to their ability to better load balance. One interesting thing to note is that, even though both `omp_dynamic` and `omp_guided` have a hard time retaining loop affinity, `omp_guided` still performed better than `omp_dynamic` on these microbenchmarks. We suspect that this is because `omp_guided` incurs lower synchronization overhead as it needs to access the centralized work queue less frequently compared to `omp_dynamic`.

Results on NAS benchmarks

Absolute performance. We ran the five NAS benchmarks on all platforms using input class size C for `mg`, `ft`, and `cg` and input class size B for `ep` and input class size D for `is`. We have used input class size B for `ep` because the benchmark is embarrassingly parallel with ample parallelism and at the same time without much data usage; as such, using the input class size C would not provide much more information compared to using class size B. For `is`, we chose to use input class size D so that the workload incurs computation that runs sufficiently long on 32 cores.

Each benchmark consists of multiple parallel loops. We manually examined the codebase to see if the parallel loops have balanced or unbalanced workloads judging by the number of memory accesses done in the loop body. For OpenMP, we used `omp_static` for balanced loops and `omp_guided` for the unbalanced loops. For FastFlow, we used the default static partitioning for balanced loops and the dynamic partitioning for the unbalanced loops. It turns out that most loops seem balanced except for one in `is` that is unbalanced.

The plots in Figure 3 show the scalability of the NAS benchmarks, namely T_1/T_P. As for the plots in Figure 1, the fist column in each plot shows the work efficiency, namely T_s/T_1. Each data point corresponding to the running times used to compute the scalability plots is the average of 10 runs with standard deviation less than 4%.

There is no one platform that performs the best across all benchmarks. Our hybrid scheme outperforms the other platforms on `ft`, `is`, and `ep`, and OpenMP outperforms the others on `mg` and `cg`. However, our hybrid scheme performs second best in `mg` and `cg` whereas, interestingly, FastFlow outperforms OpenMP on `ft`. Moreover, unlike OpenMP or FastFlow, the programmer using the hybrid scheme does not need to manually examine the code and guess as to which scheme to use, which can change depending on the input.

Retaining loop affinity. The the NAS benchmarks, we evaluate how well each strategy retains loop affinity using hardware counters to measure the numbers of memory accesses serviced at the different levels of memory hierarchy (i.e., L1, L2, L3 cache hits, and L3 misses serviced by local DRAM, remote L3 cache, and remote DRAM) using the LIKWID performance monitoring suite [31]. The specific CPUs (Intel Xeon E5-4620) used for the experiments contains a hardware bug that causes certain performance counters (specifically on L3 cache and DRAM) to under count [32] However, a software patch is available from [33] (the `latego.py` script in the repository) which mitigates, but does not completely eliminate the bug. We have applied this patch for the data shown in Figure 4. Due to the limited number of hardware counters supported on the CPUs used, these numbers are collected in two sets of runs. Within each set, a subset of hardware counters shown were collected, and the median out of three runs are used. We started the hardware count collection right before the beginning of the first top-level parallel region and stopped the collect right after the end of the last top-level parallel region. Therefore, all counts included memory accessed incurred by the scheduling code.

Note that, the absolute hardware counter numbers do not accurately reflect the relative performance of the benchmarks among different strategies, for multiple reasons. First, the performance of a benchmark running using a particular strategy is not purely a function of memory access latency; it is also a function of the scheduling overhead and idle time (i.e., time incurred by an idling core that does not have work to do). Second, the latency incurred by memory accesses are spread out among cores used but not necessarily evenly. Finally, it turns out that, code written using OpenMP actually performs more overall work across all cores because its parallel regions are always executed by all cores. That means, certain computations exist in the parallel region that could have been computed sequentially and communicated to all cores, but the code is written explicitly to incur redundant computation to avoid this

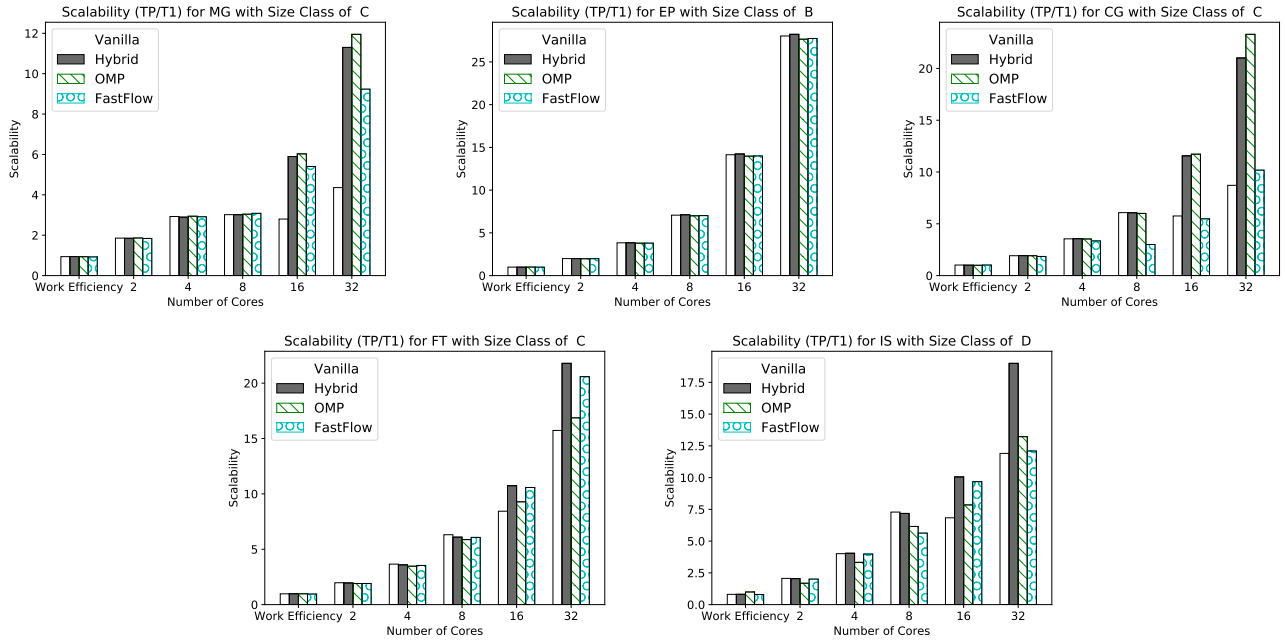

Fig. 3. The work efficiency and scalability results of the NAS benchmarks (input class size C for `mg`, `ft`, and `is` and input class size B for `ep` and `cg`).

bench	*L1*	*L2*	*local L3*	*local DRAM*	*remote L3*	*remote DRAM*	*latency w/out L1*
`hybrid mg`	1.18e11	3.42e9	5.67e8	2.73e8	1.69e7	4.41e7	1.70e11
`vanilla mg`	1.18e11	3.44e9	5.61e8	8.40e7	3.10e7	2.16e8	2.42e11
`omp mg`	2.44e11	3.50e9	5.78e8	1.75e8	3.47e6	1.06e8	1.80e11
`hybrid ep`	3.69e10	3.13e7	8.15e6	4.66e4	6.70e5	7.98e4	1.13e9
`vanilla ep`	3.69e10	3.38e7	8.17e6	4.61e4	7.48e5	8.45e4	1.20e9
`omp ep`	3.97e10	3.27e7	8.13e6	3.29e4	3.42e5	1.18e5	9.96e8
`hybrid cg`	7.92e10	1.11e10	5.46e10	2.90e9	1.90e8	2.37e8	3.36e12
`vanilla cg`	7.49e10	1.10e10	5.24e10	6.35e8	1.87e8	1.35e9	3.43e12
`omp cg`	3.13e11	1.10e10	5.34e10	1.42e9	1.11e8	1.57e9	3.77e12
`hybrid ft`	8.38e10	7.48e9	2.46e9	7.30e8	8.77e7	4.53e8	7.12e11
`vanilla ft`	8.40e10	7.58e9	2.30e9	3.10e8	1.12e8	8.54e8	8.75e11
`omp ft`	1.43e11	7.10e9	2.87e9	6.39e8	1.12e8	4.72e8	7.26e11
`hybrid is`	1.15e11	5.21e9	1.39e10	3.39e8	4.28e7	2.63e8	9.17e11
`vanilla is`	1.23e11	5.15e9	1.39e10	2.17e8	3.18e8	4.42e8	1.14e12
`omp is`	1.85e11	5.43e9	1.37e10	3.13e8	3.39e7	4.10e8	9.91e11

Fig. 4. The hardware counts for memory accesses serviced by each level of memory hierarchy running the benchmarks ion 32 cores using our scheme (`hybrid`), the dynamic partitioning in Cilk (`vanilla`), and various OpenMP schemes (`omp`), where `omp_static` was used for balanced loops and `omp_guided` was used for unbalanced loops.

Level Serviced	Latency
L1	4.1
L2	12.2
L3	41.4
local DRAM	246.7
remote L3	381.5 - 648.8
remote DRAM	643.2 - 650.9

Fig. 5. Access latency serviced by different levels of memory hierarchy on the machined used to collect data. The latency for the remote L3 and remote DRAM is given as a range because it depends on the cache line state and distance between the requesting core and where the data resides.

communication overhead. Consequently, code written using OpenMP actually incurred higher number of overall memory accesses, which mostly showed up as additional L1 hits compared to other strategies.

Nevertheless, these hardware counts can approximate how well a strategy retains loop affinity. Since the redundant computation done by OpenMP code seemed to mostly appear as additional L1 hits, we have computed the inferred latency based on these counts. Figure 5 shows the memory access latency incurred when an access is serviced at each given level of memory hierarchy on the system used to collect these numbers. When the latency is shown as a range, we used the middle value within the range. The latency measurements were collected using the Intel Memory Latency Checker [34], which was configured to access memory in a random pattern with the hardware prefetcher on. Again, we focus our attention on comparing the hybrid scheme against the dynamic partitioning in Cilk (`vanilla`) and the OpenMP schemes (`omp`) and omit FastFlow, as the schemes used by FastFlow are also supported by OpenMP, and the OpenMP schemes outperformed FastFlow for most benchmarks. As can be seem,

Based on the results shown in Figure 4, we can see that all three schemes tend to have comparable numbers for L1, L2, and L3 hits. However, as the execution starts incurring L3 misses, for `hybrid` and `omp`, these misses tend to be serviced by local DRAM more, where as `vanilla` incurred more misses serviced by remote DRAM, which backs up our intuition that the `hybrid` and `omp_static` tend to retain loop affinity better. The inferred latency (without L1) also suggests that `vanilla` tends to incur the highest latency, with the exception of `cg` which has comparable numbers across strategies.

VI. Related Work

In the literature, researchers have investigated multiple scheduling schemes for parallel loops. Beyond the various partitioning schemes mentioned in Section I, researchers have investigated in other dynamic chunking schemes that account for

load imbalance and different core starting time, making different tradeoffs between load balancing and synchronization overhead, including schemes with decreasing chunk sizes [15]–[19] and schemes that adaptively determine chunk sizes based on runtime statistics [20], [21]. These chunking strategies can be beneficial and complementary to existing methods, especially when the platform employs work sharing with a centralized queue or when the program runs on distributed memory system with distributed queues. In our scheme, since we primarily focus on shared-memory environment and load balancing is achieved via work stealing, changing the chunking size from half of what's remaining to variable size should not significantly change the performance as successful steals tend to be inexpensive. Thus, we do not focus on chunking schemes to load balance, but instead we utilizes the claiming heuristic to potentially enable better loop affinity while still allows for distributed work stealing to guarantee the execution time bound.

In distributed memory system setting, researchers have also looked into combining static partitioning and dynamic load balancing [22]–[24], where iterations are initially distributed via static partitioning which incurs little scheduling overhead. A dynamic load balancing phase can overlap with the static phase (where processors executes the iterations obtained via static partitioning). The dynamic load balancing phase requires centralized coordination and assumes that processor loads are known. In our scheme, we utilize a distributed protocol for dynamic load balancing, which is necessary to achieve the stated execution time bound.

Thoman et al. [35] propose an automatic OpenMP loop scheduling scheme that combines both static information gathered at compile time and via runtime monitoring to choose the best scheduling strategy. Specifically, at compile time, a polyhedral model [36] is used to obtain estimation of computation load of a subrange of a given loop. At runtime, input size and external workload are also collected. Based on the gathered information, the scheduler than chooses a strategy to schedule the given loop. If the loop is small (i.e., tiny workload), it is executed sequentially. If the external workload is large enough to potentially impact scheduling of this process, then a dynamic partitioning strategy is used. If the workload estimator is available, the scheduler utilizes it to generate balanced workload across available cores. Otherwise, it falls back to the dynamic scheme. Our proposed hybrid scheme can be complementary to this work, where the scheduler can utilize our hybrid scheme in place of the dynamic scheme to potentially retain loop affinity better.

Researchers have also investigated scheduling heuristics to optimize for iterative applications by incorporating affinity scheduling when assigning loop iterations [37], [38], where the iterations are assigned to processors / cores that they have affinity with. Lifflander et al. proposed a scheme to efficiently record the steal operations in a work stealing platform [39] and proposed a constrained work stealing scheme [25] to account for loop affinity by replaying a recorded steal tree or one provided by the programmer. These schemes do not focus on providing provable execution time bound and the constraint imposed by affinity scheduling can cause a processor / core to be idle despite work exists in the system.

Shiina and Taura [26] proposed a work stealing scheme in which the platform performs static partitioning of the workload based on an estimate of the workload under each task via the programmer-provided annotations, but allows for dynamic load balancing using the normal work stealing when necessary. Assuming that the programmer provided annotation is accurate, the scheme can results in deterministic partitioning of the iteration space while the partitioning should provide balanced workloads. This work requires programmer annotation, but it can be complementary to our scheme, where the programmer annotation dictates how one performs the initial static partitioning and the dynamic load balancing can play out in a semi-deterministic fashion using our claiming heuristic.

VII. Conclusion

This paper proposes a hybrid partitioning scheme for parallel loops, which combines the features of static and dynamic partitioning of loop iterations. We show that the hybrid scheme offers provable performance guarantees that are comparable to those of traditional dynamic-partitioning schemes. We also found that, in practice, the hybrid scheme performs comparably to traditional static partitioning when loop iterations have balanced workloads, but also incurs lower overhead and better scalability compared to other schemes when loop iterations have unbalanced workloads.

Acknowledgments

We thank the reviewers for their helpful comments. We also thank Justin Deters who assisted with the collection of hardware counters.

References

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS parallel benchmarks," *IJHPCA*, vol. 5, no. 3, p. 63–73, Sep. 1991.

[2] *OpenMP Application Program Interface, Version 4.0*, Available from https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, Nov. 2015.

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

[4] "Intel® Cilk™ Plus," https://www.cilkplus.org, Intel Corporation, 2013.

[5] *Intel(R) Threading Building Blocks*, Intel Corporation, 2012, available from http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *PLDI*. ACM, 1998, pp. 212–223.

[7] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson, "Programming with exceptions in JCilk," *Science of Computer Programming*, vol. 63, no. 2, pp. 147–171, Dec. 2008.

[8] C. E. Leiserson, "The Cilk++ concurrency platform," *J. Supercomputing*, vol. 51, no. 3, pp. 244–257, 2010.

[9] T. B. Schardl, I.-T. A. Lee, and C. E. Leiserson, "Brief announcement: Open Cilk," in *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures*. Vienna, Austria: ACM, 2018, p. 351–353.

[10] R. Barik, Z. Budimlić, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar, "The Habanero multicore software research project," in *OOPSLA*. Orlando, Florida, USA: ACM, 2009, pp. 735–736.

[11] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: the new adventures of old X10," in *PPPJ*, 2011, pp. 51–61.

[12] D. Lea, "A Java fork/join framework," in *ACM 2000 Conference on Java Grande*, 2000, pp. 36–43.

[13] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *OOPSLA*, 2005, pp. 519–538.

[14] M. Danelutto and M. Torquati, "Loop parallelism: A new skeleton perspective on data parallel patterns," in *PDP*. USA: IEEE Computer Society, 2014, p. 52–59.

[15] C. D. Polychronopoulos and D. J. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1425–1439, 1987.

[16] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A method for scheduling parallel loops," *Commun. ACM*, vol. 35, no. 8, p. 90–101, Aug. 1992.

[17] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," *IEEE Trans. Parallel Distributed Syst.*, vol. 4, pp. 87–98, 1993.

[18] J. Liu, V. A. Saletore, and T. G. Lewis, "Safe self-scheduling: A parallel loop scheduling scheme for shared-memory multiprocessors," *International Journal of Parallel Programming*, vol. 22, no. 6, pp. 589–616, 1994.

[19] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein, "Load-sharing in heterogeneous systems via weighted factoring," in *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*. Padua, Italy: ACM, 1996, p. 318–328.

[20] I. Banicescu and Z. Liu, "A dynamic scheduling method tuned to rate of weight changes," in *HPC*, 2000, pp. 122–129.

[21] I. Banicescu and V. Velusamy, "Performance of scheduling scientific applications with adaptive weighted factoring," in *IPDPS*. USA: IEEE Computer Society, 2001, p. 84.

[22] J. Liu and V. A. Saletore, "Self-scheduling on distributed-memory machines," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Portland, Oregon, USA: ACM, 1993, p. 814–823.

[23] O. Plata and F. F. Rivera, "Combining static and dynamic scheduling on distributed-memory multiprocessors," in *ICS*. Manchester, England: ACM, 1994, p. 186–195.

[24] Y. Yan, C. Jin, and X. Zhang, "Adaptively scheduling parallel loops in distributed shared-memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 1, pp. 70–81, 1997.

[25] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Optimizing data locality for fork/join programs using constrained work stealing," in *SC*, 2014, pp. 857–868.

[26] S. Shiina and K. Taura, "Almost deterministic work stealing," in *SC*. ACM, 2019.

[27] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *JACM*, vol. 46, no. 5, pp. 720–748, 1999.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[29] D. Griebler, J. Loff, G. Mencagli, M. Danelutto, and L. G. Fernandes, "Efficient NAS benchmark kernels with C++ parallel programming," in *PDP*, 2018, pp. 733–740.

[30] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding recursive fork-join parallelism into LLVM's intermediate representation," *ACM Trans. Parallel Comput.*, vol. 6, no. 4, Dec. 2019.

[31] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego, CA, 2010.

[32] Intel Corporation, "Intel® xeon® processor e5 product family specification update," https://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-family-spec-update.html, May 2020, accessed February 2021.

[33] A. Kleen, "pmu-tools," https://github.com/andikleen/pmu-tools, 2013, accessed May 2020.

[34] V. Viswanathan, K. Kumar, T. Willhalm, P. Lu, B. Filipiak, and S. Sakthivelu, "Intel® memory latency checker," Nov 2013. [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[35] P. Thoman, H. Jordan, S. Pellegrini, and T. Fahringer, "Automatic openmp loop scheduling: A combined compiler and runtime approach," in *Proceedings of the 8th International Workshop on OpenMP: OpenMP in a Heterogeneous World*, B. M. Chapman, F. Massaioli, M. S. Müller, and M. Rorro, Eds. Rome, Italy: Springer Berlin Heidelberg, 2012, pp. 88–101.

[36] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*. Paphos, Cyprus: Springer-Verlag, 2010, p. 283–303.

[37] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, "Locality and loop scheduling on numa multiprocessors," in *1993 International Conference on Parallel Processing - ICPP'93*, vol. 2, 1993, pp. 140–147.

[38] E. P. Markatos and T. J. LeBlanc, "Using processor affinity in loop scheduling on shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 4, pp. 379–400, 1994.

[39] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Steal tree: Low-overhead tracing of work stealing schedulers," in *PLDI*. Seattle, Washington, USA: ACM, 2013, pp. 507–518.