A NUMA-Aware Provably-Efficient Task-Parallel Platform Based on the Work-First Principle

Justin Deters Jiaye Wu Yifan Xu I-Ting Angelina Lee
Washington University in St. Louis
{j.deters, jiaye.wu, xuyifan, angelee}@wustl.edu

***Abstract*—Task parallelism is designed to simplify the task of parallel programming. When executing a task parallel program on modern NUMA architectures, it can fail to scale due to the phenomenon called *work inflation*, where the overall processing time that multiple cores spend on doing useful work is higher compared to the time required to do the same amount of work on one core, due to effects experienced only during parallel executions such as additional cache misses, remote memory accesses, and memory bandwidth issues.**

One can mitigate work inflation by co-locating the computation with its data, but this is nontrivial to do with task parallel programs. First, by design, the scheduling for task parallel programs is automated, giving the user little control over where the computation is performed. Second, the platforms tend to employ work stealing, which provides strong theoretical guarantees, but its randomized protocol for load balancing does not discern between work items that are far away versus ones that are closer.

In this work, we propose NUMA-WS, a NUMA-aware task parallel platform engineered based on the work-first principle. By abiding by the work-first principle, we are able to obtain a platform that is work efficient, provides the same theoretical guarantees as a classic work stealing scheduler, and mitigates work inflation. We have extended Cilk Plus runtime system to implemented NUMA-WS. Empirical results indicate that the NUMA-WS is work efficient and can provide better scalability by mitigating work inflation.

***Index Terms*—work stealing, work-first principle, NUMA, locality, work inflation**

I. Introduction

Modern concurrency platforms are designed to simplify the task of writing parallel programs for shared-memory parallel systems. These platforms typically employ ***task parallelism*** (sometimes referred to as ***dynamic multithreading***), in which the programmer expresses the *logical* parallelism of the computation using high-level language or library constructs and lets the underlying scheduler determine how to best handle synchronizations and load balancing. Task parallelism provides a programming model that is ***processor oblivious***, because the language constructs expose the logical parallelism within the application without specifying the number of cores on which the application will run. Examples of such platforms include OpenMP [1], Intel's Threading Building Blocks (TBB) [2], [3], various Cilk dialects [4]–[9], various Habanero dialects [10], [11], Java Fork/Join Framework [12], and IBM's X10 [13].

This research was supported in part by National Science Foundation under grant number CCF-1527692 and CCF-1733873.

Most concurrency platforms, including ones mentioned above, schedule task parallel computations using ***work stealing*** [14]–[17], a randomized distributed protocol for load balancing. Work stealing, in its classic form, provides strong theoretical guarantees. In particular, it provides asymptotically optimal execution time [14]–[17] and allows for good cache locality with respect to sequential execution when using private caches [18]. In practice, work stealing has also been demonstrated to incur little scheduling overhead and can be implemented efficiently [5].

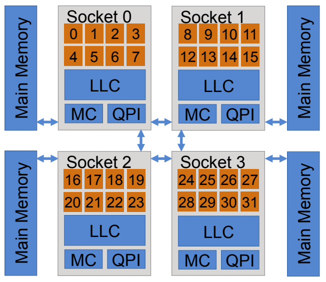

Fig. 1. An example of a 32-core four-socket system, where each socket has its own last-level L3 cache and memory banks.

Shared memory on modern parallel systems is often realized with ***Non-Uniform Memory Access (NUMA)***, where the memory latency can vary drastically, depending on where the memory access is serviced. Figure 1 shows an example of a NUMA system and its memory subsystem. This system consists of four sockets, with eight cores per socket, and each socket has its own last-level cache (LLC), memory controller, and memory banks (DRAM). Each LLC is shared among cores on the same socket, and the main memory consists of all the DRAMs across sockets, where each DRAM is responsible for a subset of the physical address range. On such a system, when a piece of data is allocated, it can either reside on physical memory managed by the ***local*** DRAM (i.e., on the same socket) or by the ***remote*** DRAM (i.e., on a different socket). When accessed, the data is brought into the local LLC and its coherence is maintained by the cache coherence protocol among LLCs. Thus, the memory access latency can be tens of cycles (serviced from the local LLC), over a hundred cycles (serviced from a local DRAM or a remote LLC), or a few hundreds of cycles (serviced from a remote DRAM).

A task parallel program can fail to scale on such a NUMA system, as a result of a phenomenon called ***work inflation***, where the overall processing time that multiple cores spend on doing useful work is much higher compared to the time required to do the same amount of work on one core, due to effects experienced only during parallel executions. Multiple factors can contribute to work inflation, including work migration, parallel computations sharing a LLC destructively, or accessing data allocated on remote sockets.

One can mitigate work inflation by co-locating computations that share the same data or co-locate the computation and its data on the same socket, thereby reducing remote memory accesses. These strategies are not straightforward to implement in task parallel code scheduled using work stealing, however. First, by design the scheduling of task parallel programs is automated, which gives the programmer little control over where the computation is executed. Second, the randomized protocol in work stealing does not discern between work items that are far away versus ones that are closer.

Ideally, we would like a task parallel platform that satisfies the following criteria:

- provide the same strong theoretical guarantees that a classic work stealing scheduler enjoys;
- be ***work efficient***, namely, the platform does not unnecessarily incur scheduling overhead that causes the single-core execution time to increase;
- support a similar processor-oblivious model of computation: assuming sufficient parallelism, the same program for a given input should scale as the number of cores used increases; and
- mitigate work inflation.

Even though many mechanisms and scheduling policies have been proposed to mitigate work inflation in task parallel programs [19]–[29], none of the proposed solutions satisfy all criteria simultaneously. In particular, many of them are not work efficient nor do they provide a provably efficient scheduling bound (see Section VI).

In this paper, we propose NUMA-WS, a task parallel platform that satisfies these criteria simultaneously. NUMA-WS employs a variant of work stealing scheduler that extends the classic algorithm with mechanisms to mitigate NUMA effects. NUMA-WS achieves the same theoretical bounds on execution time and additional cache misses for private caches as the classic work stealing algorithm, albeit with a slightly larger constant hidden in the big-O term.

NUMA-WS provides the same execution time bound as classic work stealing [14]–[17], which can be quantified using two important metrics: the ***work***, as defined by the execution time running the computation on one core, and the ***span***, the longest sequential dependences in the computation, or its theoretical running time executing on infinitely-many cores. Given a computation with T_1 ***work*** and T_∞ ***span***, NUMA-WS executes the computation on P cores in expected time $T_1/P + O(T_\infty)$.[1] The additional cache misses due to parallel execution are directly correlated with the number of times computation "migrates," or when the order of computation during parallel execution diverges from that of a single-core execution. In NUMA-WS, the number of times such divergence can occur is upper bounded by $O(PT_\infty)$, same as with classic work stealing [18].

[1]Even without accounting for scheduling overhead, this is the best bound possible when the dependences of the parallel computation are not unknown until execution time [30], [31].

To measure work efficiency, an important metric is T_S, the execution time of ***serial elision***, obtained by removing the parallel control constructs or replacing it with its sequential counter part. Serial elision should perform the exact same algorithm that the parallel program implements but without the parallel overhead. Thus, one can quantify work efficiency by comparing T_1 against T_S to measure parallel overhead. Assuming a work-efficient platform, one can obtain parallel code whose ratio between T_1 and T_S is close to one.

We have implemented a prototype system by extending Intel Cilk Plus, which implements the classic work stealing algorithm, and empirically evaluated it. The empirical results indicate that NUMA-WS is work efficient, scalable across different number of cores, and can mitigate NUMA effects. Specifically, we show that, NUMA-WS incurs negligible parallel overhead (i.e., T_1/T_S), comparable to that in Cilk Plus. Moreover, we compare the parallel execution times across benchmarks when running on Intel Cilk Plus versus running on NUMA-WS on a four-socket 32-core system. NUMA-WS was able to decrease work inflation compared to Cilk Plus without adversely impacting scheduling time. Across benchmarks, NUMA-WS obtains much better speedup than that in Cilk Plus.

Critically, to achieve the theoretical bounds and practical efficiency, the design and engineering of NUMA-WS abides by a principle called the ***work-first principle*** proposed by [5], which states that one should minimize the overhead borne by the work term (T_1) and move the overhead onto the span term (T_∞). Intuitively, a scheduler must incur some scheduling overhead due to the extra bookkeeping necessary to enable correct parallel execution or to mitigate NUMA effects. Within the context of a work-stealing scheduler, worker threads (surrogates of processing cores) load balance by "stealing" work when necessary. The work-first principle states that it's best to incur scheduling overhead on the control path that can be amortized against successful steals. To put it differently, whenever a choice can be made to incur overhead on a thief stealing versus on a worker busy working, it's always preferred to incur the overhead on the thief stealing.

Contributions

To summarize, this paper makes the following contributions:

- We present NUMA-WS, a NUMA-aware task parallel platform that implements a work stealing algorithm with mechanisms to mitigate work inflation (Section III).
- We show that our extended work stealing algorithm retains the same theoretical guarantees on execution time and cache bounds as the classic algorithm (Section IV).
- We implemented and empirically evaluated NUMA-WS. The empirical results show that NUMA-WS is work efficient, scalable across cores, and mitigates work inflation (Section V).

II. Preliminaries: Work Stealing in Cilk Plus

Our prototype implementation of NUMA-WS extends Intel Cilk Plus [9], which implements the classic work stealing algorithm. The engineering of Cilk Plus also follows the work-first principle. In this section, we review the implementation of Cilk Plus and examine its work efficiency to demonstrate the benefit of the work-first principle. Next, we examine the work inflation of multiple benchmarks running on Cilk Plus to motivate the need for a NUMA-aware work-efficient runtime.

The language model. Cilk Plus extends C/C++ with two parallel primitives: `cilk_spawn` and `cilk_sync`.[2] When a function F ***spawns*** another function G (invoking G with the keyword `cilk_spawn`), the ***continuation*** of F, i.e., statements after the `cilk_spawn` call, may execute in parallel with G. The keyword `cilk_sync` specifies that control cannot pass beyond the `cilk_sync` statement until all previously spawned children within the enclosing function have returned.

These keywords denote the *logical* parallelism of the computation. When F spawns G, G may or may not execute in parallel with the continuation of F, depending on the hardware resource available during execution.

Work stealing and the work-first principle. In work stealing, each ***worker*** (a surrogate of a processing core) maintains a ***deque*** (a double ended queue) of work items. Each worker operates on its own deque locally most of the time and communicates with one another only when it runs out of work to do, i.e., its deque becomes empty. When that happens, a worker turns into a ***thief*** and *randomly* chooses another worker, the ***victim***, to steal work from. A worker, while busy working, always operates at the tail of its own deque like a stack (i.e., first in last out). A thief, when stealing, always steals from the head of a victim's deque (i.e., taking the oldest item).

The ***work-first*** principle [5] states that one should minimize the overhead borne by the work term (T_1) and move the overhead onto the span term (T_∞), which corresponds to the steal path. A work-stealing runtime abiding by the work-first principle tends to be work efficient, as demonstrated by the implementation of Cilk-5. Subsequent variants of Cilk [6]–[8] including Cilk Plus follow similar design.

The intuition behind the work-first principle can be understood as follows. The parallelism of an application is defined as T_1/T_∞, or how much work there is along each step of the span. Assuming the application contains ample parallelism, i.e., $T_1/T_\infty \gg P$, the execution time is dominated by the T_1/P term, and thus it's better to incur overhead on the T_∞ term. Moreover, in practice, when the application contains ample parallelism, steals occur infrequently.

Work stealing in Cilk Plus. Figure 2 shows the pseudocode for the work-stealing scheduler in Cilk Plus. Note that when no steal occurs, the one-worker execution follows that of the serial elision. Upon a `cilk_spawn`, the worker pushes the *continuation* of the spawning parent at the tail of its deque (line 1) and continues to execute the spawned child (line 2), which can also spawn. Once pushed, the continuation of the parent becomes stealable. Upon returning from a `cilk_spawn`, the

```
F spawns G:
 1  PUSHDEQUEATTAIL(F); // F's continuation becomes stealable
 2  continue to execute G

G returns to its spawning parent F:
 3  success = POPDEQUEATTAIL();
 4  if success // F is not stolen and must be at the tail of the deque
 5      continue to execute F
 6  else // parent stolen; the deque is empty
 7      next_action = CHECK_PARENT
 8      return to scheduling loop

F executes cilk_sync:
 9  if F.stolen = TRUE
10      // F must be a full frame and the deque is empty
11      success = CHECKSYNC(); // must do a nontrivial sync
12      if success
13          continue to execute F
14      else // F must be the only thing in the deque
15          suspend F
16          next_action = STEAL
17          return to scheduling loop
18  else continue to execute F // nothing else needs to be done

scheduling loop: // frame is a either NULL or the first root full frame
19  while computation-done = FALSE
20      if next_action = CHECK_PARENT
21          frame = CHECKPARENT();
22          next_action = STEAL // reset next_action
23      if frame = NULL
24          frame = RANDOMSTEAL();
25      else RESUME(frame)
```

Fig. 2. Pseudocode for the Cilk Plus work-stealing scheduler: when a function spawns, when a spawned function returns, when a function executes `cilk_sync`, and its scheduling loop. Here, we use F to represent both a function instance and its corresponding frame. The variable $next\_action$ specifies what the scheduling loop should do next.

worker pops the parent off the tail of its deque (if not stolen) to resume its execution (lines 3–5).

The strategy of pushing the continuation of the parent is called ***continuation-stealing***. An alternative implementation is to push the spawned child, called ***child-stealing***.[3] Cilk Plus implements continuation-stealing because it can be more space efficient; more importantly, it allows a worker's execution between successful steals to mirror exactly that of the serial elision. Thus, one can optimize the cache behavior of parallel code for private caches by optimizing that of the serial elision.

Runtime organization based on the work-first principle. Two aspects of the Cilk Plus design follow from the work-first principle: the "THE protocol," proposed by [5] and the organization of the runtime data structures, as described in [32]. The THE protocol is designed to minimize the overhead of a worker operating on its deque, allowing a victim who is doing work to not synchronize with a thief unless they are both going after the same work item in the deque. The THE protocol remains unchanged in NUMA-WS and thus we omit the details here and refer interested readers to [5]. We briefly review the organization of the runtime data structures, which is most relevant to the design of NUMA-WS.

[2]A third keyword `cilk_for` exists, which specifies that the iterations for a given loop can be executed in parallel; it is syntactic sugar that compiles down to binary spawning of iterations using `cilk_spawn` and `cilk_sync`. Other concurrency platforms contain similar constructs with similar semantics, though the syntax may differ slightly.

[3]In the literature, continuation-stealing is sometimes referred to as ***work-first*** and child-stealing referred to as ***help-first***. The choice of which strategy to implement is orthogonal to the work-first principle. Hence, we call them "continuation" versus "child-stealing" here to avoid confusion.

The runtime data structures in Cilk Plus are organized around the work-first principle, so as to incur as little overhead on the work path as possible, at the expense of incurring overhead on the steal path. With continuation-stealing, in the absence of any steals, the behavior of a worker mirrors exactly that of the serial elision, and the execution should incur little scheduling overhead. On the other hand, when a successful steal occurs, *actual* parallelism is realized, because a successful steal enables the continuation of a spawned parent to execute concurrently (on the thief) with its spawned child (on the victim). In this case, the runtime must perform additional bookkeeping in order to keep track of actual parallel execution.

In Cilk Plus, a ***Cilk function*** that contains parallel keywords is treated as an unit of scheduling: every Cilk function has an associated ***shadow frame*** that gets pushed onto the deque upon spawning. It is designed to be light weight, storing the minimum amount of information necessary in order to enable parallel execution (i.e., which continuation to resume next). Whenever a frame is stolen successfully, however, the runtime promotes the stolen frame from a shadow frame into a ***full frame*** which contains the necessary bookkeeping information to keep track of actual parallel execution.

That means, only a full frame can have spawned children executing concurrently. Thus, a frame's stolen field (e.g., $F.\mathit{stolen}$) is only ever set for a full frame that has been stolen but has not executed a `cilk_sync`. Execution of a `cilk_sync` checks for the flag, and only if the flag is set, then a ***nontrivial sync*** needs to be invoked that checks for outstanding spawned children executing on other workers (line 11). On the other hand, for a shadow frame, its flag is never set and executing a `cilk_sync` is a no-op, as its corresponding function cannot have outstanding spawned children and thus nothing needs to be done (line 18).

If a nontrivial sync is necessary, and there are outstanding spawned children executing on other workers, then the current worker suspends this frame and returns to the runtime to find more work to do (i.e., steal) (lines 15–17). The suspended frame then becomes the responsibility of the worker who executes the last spawned child returning. Thus, a child returning from a spawn, upon detecting that its parent has been stolen, returns back to the scheduling loop (lines 6–8) and checks if its parent is ready to resume (lines 20–22), i.e., it is the last spawned child returning.

Since work stealing always steals from the head of the deque, when a worker is about to return control back to the scheduling loop (lines 8 and 17), its deque must be empty. Thus, the scheduling loop handles only full frames. Upon returning to the loop, a worker executes the loop repeatedly until the computation is done or some work is found, either via resuming a suspended parent (line 21) or via successful steals (line 24).

Work efficiency and work inflation of Cilk Plus. The time a worker spends can be categorized into three categories — ***work time***, time spent doing useful work (i.e., processing the computation), ***idle time***, time spent trying to steal but failing to find work to do, and ***scheduling time***, time spent performing scheduling related tasks to manage actual parallelism, such as frame promotions upon successful steals and nontrivial syncs. By looking at how much work time that all workers collectively spend grows as the number of cores increases, one can gauge how much work inflation impacts the scalability of the program. On the other hand, idle time is a good indication of how much parallelism a computation has — a computation that does not have sufficient parallelism is generally marked by high idle time. Finally, scheduling time indicates the runtime overhead. See [33] for the formalization of this intuition.

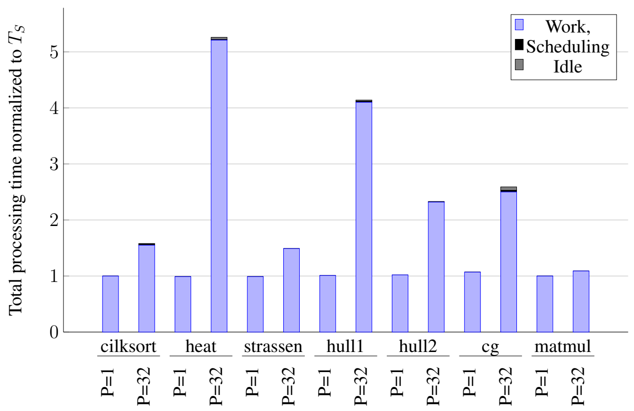

Fig. 3. The normalized total processing times of benchmarks running on Cilk Plus, normalized to T_S, the execution time of the corresponding serial elision. The `P=1` bars show the normalized total processing times running on one core; the `P=32` bars show the normalized total processing times running on 32 cores. For `P=32`, each data point is also broken down into three categories: work time, scheduling time, and idle time. We have two data sets with different characteristics for `Hull` and thus two sets of data points are shown for `Hull`.

Figure 3 shows the normalized total processing times of six benchmarks running on Cilk Plus, normalized to T_S, the execution time of the corresponding serial elision.[4] Note that `P=1` is simply T_1, the time running on one worker, which includes the spawn overhead, but not scheduling or idle time, since no actual parallelism is realized executing on one worker.

Cilk Plus has high work efficiency, as the ratio between T_1 and T_S is close to one. Here, we did coarsen the base case — when a divide-and-conquer parallel algorithm reaches certain base case size, the code stops spawning and calls the sequential version instead. Coarsening helps with spawn overhead and is a common practice for task-parallel code; it makes a trade-off between spawn overhead and parallelism — the smaller the base case size the higher the parallelism and spawn overhead. Typically one can coarsen the base case size to mitigate spawn overhead but keeps it small enough so that the computation has sufficient parallelism. We chose the base case sizes for these benchmarks by picking the ones that provide the best T_{32} raw execution times.

For `P=32`, most benchmarks have very little scheduling time and idle time, indicating that the scheduler is efficient and there is sufficient parallelism to saturate 32 cores. Finally, we look at work inflation. Benchmarks tested have work inflation ranging

[4]The data is collected using the same setup including input and base case sizes described in Section V.

from $1.45\times$ to $5.24\times$, with the exception of `matmul`.[5] The implementation of `matmul` is already cache oblivious [34] and thus had little work inflation to begin with. We use this benchmark as a baseline to show that, even for a benchmark does not really benefit from NUMA-aware scheduling, the additional scheduling mechanism in NUMA-WS does not adversely impact performance. Moreover, the data layout transformation helps when applied to `malmul`.

III. NUMA-Aware Task-Parallel Platform

NUMA-WS extends Cilk Plus to incorporate mechanisms to mitigate NUMA effects. An effective way to mitigate NUMA effects is to co-locate computations that share data, and/or co-locate data and the computation that uses the data. To that end, NUMA-WS includes user-level APIs to provide locality hints, extends the work stealing scheduler to be NUMA-aware, and schedules computations according to the locality hints when possible. In addition, NUMA-WS also includes a simple data layout transformation that is applied to a subset of benchmarks. This section describes these extensions in detail.

A. Modifications to the Programming Model

To facilitate NUMA awareness, the runtime at startup ensures that the worker-thread-to-core affinity is fixed. Since the programming model is processor-oblivious, an application can run on any number of cores and sockets within the constraints of the hardware resource. We assume that the user decides how many cores and how many sockets an application runs on when invoking the application but does not change this configuration dynamically at runtime.

Given the user-specified number of cores and sockets, the runtime spreads out the worker threads evenly across the sockets and groups the threads on a given socket into a single group. Each group forms a ***virtual place*** that forms the basic unit for specifying locality.

The locality API allows the user to query the number of virtual places in the application code and specify which virtual place a spawned subcomputation should ideally run on. If the user specifies the locality for a spawned subcomputation G, by default any computation subsequently spawned by G is also marked to have the same locality. Such a default works well for recursive divide-and-conquer algorithms, which task parallelism is well-suited for. For the benchmarks tested, this default means that only the top-level root Cilk function needs to specify locality hints. The API also includes ways to unset or update the locality hints for a Cilk function.

Figure 4 shows how one might use locality hints in a parallel mergesort, where MergeSortTop is the top-level root Cilk function: it recursively sorts the four quarters of the input array in place, and uses the temporary array to merge the sorted quarters back into the input array. The locality hints are specified using the `@p#` notation, where a `p#` is a variable storing the ID of a virtual place that the task (the corresponding function call and its subcomputation) should execute.

[5]The precise numbers for all benchmarks are shown in Section V.

```
MergeSortTop(int *$in$, int *$tmp$, int $n$)
if $n <$ BASE_CASE
    QuickSort($in$, $n$); // in-place sequential sort
else
    // initialized p0, p1, p2, and p3 based on number of places
    // $in$ and $tmp$ are partitioned appropriately.
    cilk_spawn MergeSort($in$, $tmp$, $n/4$) // implicitly @p0
    cilk_spawn MergeSort($in$ + $n/4$, $tmp$ + $n/4$, $n/4$); @p1
    cilk_spawn MergeSort($in$ + $n/2$, $tmp$ + $n/2$, $n/4$); @p2
    MergeSort($in$ + $3n/4$, $tmp$ + $3n/4$, $n$ - $3n/4$); @p3
    cilk_sync;
    int *$tmp1$ = $tmp$, *$tmp2$ = $tmp$ + $n/2$;
    cilk_spawn ParMerge($in$, $in$ + $n/4$, $n/4$, $n/4$, $tmp1$); @p0
    ParMerge($in$ + $n/2$, $in$ + $3n/4$, $n/4$, $n$ - $3n/4$, $tmp2$); @p2
    cilk_sync;
    ParMerge($tmp1$, $tmp2$, $n/2$, $n$ - $n/2$, $in$); @ANY
```

Fig. 4. Pseudocode for the top-level function for parallel mergesort with locality hints (as denoted by @p# or @ANY). The MergeSort is defined similarly as the MergeSortTop but without the locality hint: it takes in an unsorted input array, a temporary array, and their sizes, and sorts the input array in place. The ParMerge performs parallel merge; it takes in two sorted input arrays, their sizes, and an output array, and merges the two sorted inputs into the output. Variables p0–p3 store IDs for virtual places based on the number of sockets used, but they do not have to be distinct (e.g., if less than four sockets are used). The ANY indicates no place constraints and unsets the locality hint.

Assuming the computation runs on four sockets, the code would initialize each of the virtual places to the appropriate socket, and specify that the i^{th} quarter should be sorted at the i^{th} virtual place. For merge, since we have to merge two arrays sorted at two different places together, we simply specify its locality to be one of the virtual places that the inputs come from. Even though we have only specified work for two virtual places for the merge phase, the place specifications are only hints, and the NUMA-WS runtime will load balance work across all sockets dynamically as necessary.

With continuation-stealing, the first spawned child is always executed by the same worker who executes the corresponding `cilk_spawn`, and thus we do not specify a locality hint for the first spawn. By default, the runtime always pins the worker who started the root computation at the first core on the first socket and thus implicitly the first spawned child always executes at the first virtual place (at `p0` in the code). If the user had specified a locality hint for the first spawned child that differs from where the parent is executing, the spawned child will obtain the user-specified locality, but will not get moved to the specified socket immediately. Rather, the computation may get moved later as steals occur according to the lazy work pushing mechanism described in Section III-B.

In order to benefit from the locality hints, one should allocate the data on the same socket that the computation belongs to. In this example, one should allocate the physical pages mapped in the i^{th} quarters of the in and tmp arrays from the socket corresponding to the i^{th} virtual place. We have developed library functions that allow the application code to do this easily at memory allocation time, but they are simply accomplished by calling the underlying `mmap` and `mbind` system calls that Linux OS provides.

The API described is an idealized API, which requires compiler support. In our current implementation, we manually hand compiled the application code by explicitly invoking runtime functions to indicate the place specifications. The transformation from the idealized API to the runtime calls is quite mechanical and can easily be done by a compiler.

B. Modifications to the Scheduler

The locality specified is only a hint, and the runtime tries to honor it with best effort and do so in a work-efficient fashion, incurring overhead only on the span term. Note that enforcing locality strictly can impede runtime's ability to load balance, e.g., the computation contains sufficient parallelism overall but most parallelism comes from computation designated for a single socket. Thus, the NUMA-WS runtime may execute the computation on a different socket against the locality hint if doing so turns out to be necessary for load balancing.

To obtain benefit using NUMA-WS, we expect the application code to perform data partitioning with the appropriate locality hints specified. However, because the runtime is designed to foremost treat load balancing as the first priority and account for locality hints with best effort, not specifying locality hints would not hurt performance much and result in comparable performance with one obtainable with the Cilk Plus runtime.

At a high level, NUMA-WS extends the existing work-stealing scheduler with two NUMA-aware mechanisms:

- **Locality-biased steals:** In the classic work stealing algorithm, when a thief steals, it chooses a victim uniformly at random, meaning that each worker gets picked in with $1/P$ probability (and if the worker happens to pick itself, it tries again). The locality-biased steals simply change the probability distribution of how a worker steals, biasing it to preferentially steal work from victims running on the (same) local socket over victims on the remote sockets.
- **Lazy work pushing:** The ***work pushing*** refers to the operation that, a worker, upon receiving a work item, instead of executing the work, pushes it to a different worker to honor the locality hint. Work pushing has been proposed in prior work (see Section VI). However, the key distinction between our work and prior work is that, NUMA-WS performs ***lazy work pushing***, in a way that abides by the work-first principle and incurs overhead only on the span term.

Figure 5 shows NUMA-WS's modifications to work stealing to incorporate these mechanisms, which we explain in detail next. Compared to the Cilk Plus scheduler, NUMA-WS may perform additional operations when executing a nontrivial sync (lines 5–11), inside the scheduling loop (lines 21–26), and it uses a modified stealing protocol (line 28). The operations performed on a spawn and a spawn returning are the same as shown in Figure 2, so we do not repeat them here.

If the runtime performs work pushing indiscriminately, the overhead incurred by work pushing would be on the work term instead of on the span term. Such overhead includes doing extra operations that do not advance towards finishing the computation and synchronizing with the ***receiving*** worker (i.e., worker to push work to), which would result a work inefficient runtime.

```
F executes sync:
 1  if $F.stolen$ = TRUE
 2      // $F$ must be a full frame and the deque is empty
 3      $success$ = CHECKSYNC(); // must do a nontrivial sync
 4      if $success$
 5          if $F.place \neq worker.place$
 6              if PUSHBACK($F$) = TRUE
 7                  // another worker will resume $F$
 8                  $next\_action$ = STEAL
 9                  return to scheduling loop
10              else continue to execute $F$
11          else continue to execute $F$
12      else
13          suspend $F$
14          $next\_action$ = STEAL
15          return to scheduling loop
16  else continue to execute $F$ // nothing else needs to be done
```

```
scheduling loop: // $frame$ is a either NULL or the first root full frame
17  while $computation\text{-}done$ = FALSE
18      if $next\_action$ = CHECK_PARENT
19          $frame$ = CHECKPARENT();
20          $next\_action$ = STEAL // reset $next\_action$
21          if $frame \neq$ NULL and $frame.place \neq worker.place$
22              if PUSHBACK($frame$) = TRUE
23                  // Another worker will resume $frame$
24                  $frame$ = NULL
25      if $frame$ = NULL
26          $frame$ = POPMAILBOX();
27      if $frame$ = NULL
28          $frame$ = BIASEDSTEALWITHPUSH();
29      else RESUME($frame$)
```

Fig. 5. Modifications to the work-stealing scheduler in NUMA-WS. Here, PUSHBACK() is the mechanism where the runtime pushes a full frame to the virtual place that it has been designated for. $F.place$ stores the locality hint for F and $worker.place$ stores the virtual place that the worker belongs to.

Instead in NUMA-WS, work pushing only occurs when a worker is handling a full frame. Recall from Section II that, the data structures in the Cilk Plus runtime are organized in accordance to the work-first principle: a frame is either a shadow frame that has never been stolen before, or a full frame that has been stolen successfully in the past and can contain actual parallelism underneath (i.e., may have outstanding spawned children executing on different workers). Specifically, a worker only performs work-pushing in the following scenarios:

- A worker executes a non-trivial `cilk_sync` successfully, and the synched full frame is earmarked for a different socket via its locality hint (lines 5–11).
- A worker returns from a spawned child, whose parent had executed a nontrivial `cilk_sync` unsuccessfully and therefore is suspended. The returning child is the last spawned child returning, and thus the parent is ready to be resumed at the continuation of `cilk_sync`, but the parent is earmarked for a different socket (lines 21–24).
- A thief steals successfully, and the stolen full frame is earmarked for a different socket (part of BIASEDSTEALWITHPUSH in line 28).

Besides judiciously selecting certain control paths to perform work pushing, another crucial aspect of lazy work pushing is how it selects the receiving worker and how to push work without interrupting the receiving worker. This logic is implemented in PUSHBACK, which we explain below.

When a worker needs to push a full frame to a designated socket, it *randomly* chooses a receiving worker on the designated socket to push the frame to. The chosen receiving worker may be busy working and likely have a non-empty deque. Thus, each worker besides managing a deque, also has a ***mailbox***, that allows a different worker to deposit work designated for the receiving worker without interrupting. Crucially, the mailbox contains only one entry; that is, each worker can have only one single outstanding ready full frame that it is not actively working on. A pushing worker may fail to push work because the randomly chosen receiving worker is busy, and has a full mailbox. If the push fails, the pushing worker increments a counter on the frame and tries again with a different randomly chosen receiving worker. Once the counter on the frame exceeds the ***pushing threshold*** (a configurable runtime parameter), the pushing worker simply takes the full frame and resumes it itself. We shall see in the analysis (Section IV) 1) how we can amortize the cost of pushing against the span term, 2) why it's crucial to have only a single entry for the mailbox, and 3) why there must be a *constant* pushing threshold.

Besides potentially calling PUSHBACK, the steal protocol implemented by BIASEDSTEALWITHPUSH (Figure 5: line 28) differs from the original RANDOMSTEAL (Figure 2: line 24) in the following ways.

First, the protocol implements locality biased steals. Given the set of virtual places, the runtime configures the steal probability distribution according to the distances between virtual places, where the distances are determined by the output from `numactl`. For instance, on a 32-core 4-socket machine shown in Figure 1, each socket i forms a virtual place i. When a worker on socket 0 runs out of work, it will preferentially select victims from the local socket (socket 0) with the highest probability, followed by victims from sockets that are one-hop away (i.e., sockets 1 and 2) with medium probability, followed by victims from the socket that is two-hop away (i.e., socket 3) with the lowest probability.

Second, since now a victim may potentially have a resumable frame in the mailbox also (besides what's in the deque), a thief stealing needs to check for both, but in a way that still retains the theoretical bounds. Specifically, when a thief steals into a victim, it will flip a coin. If the coin comes up heads, it does the usual steal protocol by taking the frame at the head of the deque (promoting it into a full frame). If the coin comes up tails, however, it checks the mailbox, followed by three possible outcomes: 1) mailbox empty, and thus the thief falls back to stealing from the deque; 2) mailbox is full, and the frame is earmarked for socket that the thief is on, so the thief takes it; 3) mailbox is full, and the frame is earmarked for a different socket; the thief then calls PUSHBACK and performs the lazy work pushing as described before until the frame reaches the pushing threshold (in which case the thief can simply take it).

We shall see in the analysis (Section IV) 1) why the coin flip is necessary and 2) why this biased steal protocol still provides provable guarantees.

Finally, given that each worker now has a mailbox potentially containing work, another small modification to the scheduling loop is needed. When a worker runs out of work and returns to the scheduling loop (in the $next\_action$ = STEAL case), it checks if something is in its mailbox first (Figure 5, line 26). If so, it simply resumes it next. Note that if a worker is back to the scheduling loop with its $next\_action$ set to STEAL, its deque must be empty.

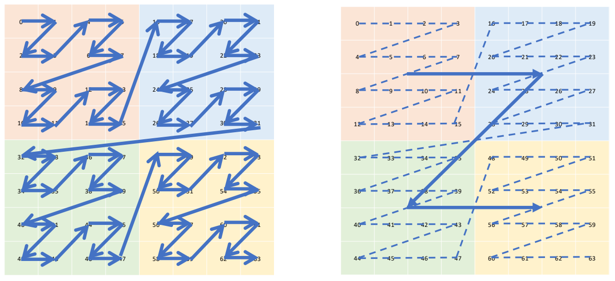
(a) Z-Morton **(b) blocked Z-Morton**

Fig. 6. **(a)** Ordinary Z-Morton layout on the cell-by-cell basis where every element lies on a recursive Z curve. **(b)** The blocked Z-Morton layout used in our data layout transformation, where the blocks are laid out recursively on the Z curve, and data within each block is laid out in row-major order.

C. Simple Data Layout Transformation

An effective way to mitigate work inflation due to NUMA is to co-locate data and computation that uses the data. Even though modern OSs tend to provide facility to bind pages to specific sockets, one must specify data allocation in page granularity. For applications that operate on 2D arrays such as various matrix operations, the usual row-major order data layout is not conducive to data and computation co-location, since most parallel algorithms use divide-and-concur techniques which recursively subdivide the data into smaller pieces that can span multiple rows but only part of the rows. Thus, by the time we reach the base case of a divide-and-conquer algorithm, the base case would be accessing data that is scattered across multiple physical pages, making it challenging to co-locate data and the computation.

One way to fix it is to use ***Z-Morton*** layout (also called ***cache-oblivious bit-interleaved*** layout), which interleaves the bits when calculating the index so that the data is laid out in a recursive Z curve [34] (Figure 6a). Computing indices for Z-Morton layout on the cell-by-cell basis is costly, however.

To achieve co-location, our platform provides APIs to allow the user to perform data layout transformation. The data layout transformation simply lays out blocks in Z-Morton layout and lays out the data within each block in row-major order, as shown in Figure 6b. Doing so has the following benefits: 1) the data accessed by the base case of a divide-and-conquer algorithm that utilizes 2D arrays will be contiguous in memory; and 2) since the bit interleaving needs to be computed only for the block indices, we save on overhead for index computation. While the use of Z-Morton layout is not new, in Section V, we show that the blocked Z-Morton layout works well with existing NUMA systems.

IV. NUMA-WS's Theoretical Guarantees

NUMA-WS provides the same execution time and cache miss bounds as classic work stealing. Specifically, NUMA-WS executes a computation with T_1 work and T_∞ span in expected time $T_1/P + O(T_\infty)$ on P cores, and the number of successful steals can be bounded by $O(PT_\infty)$, although the constant in front of the T_∞ term would be larger than that for classic work stealing. The bound on the number of steals allows one to constrain the additional cache misses with respect to sequential execution for private caches [18]. The structure of the theoretical analysis largely follows that described by Arora et. al [16] but with some delta. We provide the high level intuitions and describe the delta here.

We first explain the intuition behind the original work stealing analysis using the potential function formulation by Arora et. al [16], henceforth referred to as the ABP analysis. Here, we assume a dedicated execution environment.[6]

In the ABP analysis, a computation (a program for a given input) is modeled as a ***directed acyclic graph (dag)***, where each node represents a ***strand***, a sequence of instructions that contain no parallel control that must be executed by a single worker, and each edge represents a dependence — if there is an edge between node u and v, then v cannot execute until u finishes. Based on this model, a `cilk_spawn` generates a node with two out-degree, one to the spawned child and one to the continuation; a `cilk_sync` generates a node with multiple in-degree, one from each spawned child joined by the `cilk_sync`. Assuming each node takes a unit time step to compute (if a strand takes more than one time step to compute, it is split into a chain of nodes), the work is then defined as the total number of nodes in the dag, and span is the number of nodes along a longest path in the dag.

At each time step, a worker is either doing useful work, i.e., executing a node (call it a ***work step***), or stealing including failed steal attempts (call it a ***steal step***). The total number of work steps across all workers is bounded by T_1, because if the workers have collectively spent T_1 time steps doing useful work, the computation would be done. Thus, the key is to bound the number of steal steps. As long as we can bound the number of steal steps to $O(PT_\infty)$, the bound follows, since the execution time on P workers is the number of total steps divided by P.

Informally, the intuition of the analysis is as follows. The non-empty deques contain nodes ready to be executed; in particular, there is some node that is "critical" — a node on the span and if executed the span will decrease by one. Due to how work stealing operates, such a critical node must be at the head of some deque. Thus, after $O(P)$ steal attempts (since each deque has $1/P$ probability to be stolen from), the critical node is likely to be executed. Thus, after $O(PT_\infty)$ steal attempts, we exhaust the span, thereby bounding the steal steps.

To show this formally, the ABP analysis uses a potential function formulation to bound the total number of steal steps. The nodes pushed onto a worker's deque are all ready to be executed and has certain amount of potential, defined as a function of both the span of the computation and the "depth" of the node in the dag — roughly speaking, one can think of the potential function as, "how far away the node is from the end of the computation." Due to the way work-stealing operates, it maintains a property called the ***top-heavy deques***[7] (Lemma 6 in [16]): the node at the head of a non-empty deque constitutes a constant fraction of the overall potential of the deque. Moreover, after $O(P)$ steal attempts, with constant probability, the overall potential of the computation decreases by a constant fraction (Lemmas 7 and 8 in [16]). The intuition is that, after that many steals, the critical node at the head of some deque are likely to get stolen and executed, thereby decreasing the potential by a constant fraction. Since the computation started out with certain amount of potential, defined in terms of the span of the computation, there cannot be more than $O(PT_\infty)$ steal attempts before the potential reaches 0 (i.e., the computation ends).

There are two key elements in this argument. First, if a deque is non-empty, the head of a deque contains a constant fraction of the potential in the deque. Second, after $O(P)$ steal attempts, the critical node at the head of some deque gets executed.

The property of top-heavy deques still holds in NUMA-WS, because a worker still pushes to the tail and a thief steals form the head. In NUMA-WS, however, 1) the steal probability has changed (no longer uniformly at random); 2) a thief randomly chooses between the victim's mailbox and its deque; 3) even if a thief steals successfully, it can push the stolen frame back to the designated socket (into a mailbox) instead of executing it immediately. We first argue why these changes do not jeopardize the analysis. Then we show how we bound the additional cost of work pushing.

It turns out that, as long as the critical node gets stolen and executed with probability $1/(cP)$ for some non-zero constant c, we can show that $O(P)$ steal attempts cause the overall potential to decrease by a constant fraction, which in turn bounds the number of steals to $O(PT_\infty)$. In NUMA-WS, the critical node could either be at the head of some deque or in some worker's mailbox (via lazy work pushing).

Specifically, the following lemma is a straightforward generalization of Lemmas 7 and 8 in [16]:

Lemma 1. *Let $\Phi(t)$ denote the overall potential of the computation at time step t. Assuming the probability of each deque being the target of a steal attempt is at least $1/X$, then after X steal attempts, the potential is at most $\Phi(t)/4$ is at least $1/4$.*

Here, let $1/X$ be $1/(2cP)$ for some constant $c > 0$ that corresponds to the probability of stealing into a worker on the most remote socket. The factor of 2 is due to the fact that a thief only steals into a victim's deque with $1/2$ probability once a given victim is chosen (the coin flip that decides whether to steal into the deque or the mailbox).

[6]The ABP analysis provides bounds for both dedicated and multiprogrammed environments.

[7]In the ABP analysis [16], they refer to the head of the deque as its "top" and the tail as its "bottom," and hence the name of the lemma.

The rest of the proof in the ABP analysis follows, and one can show that the number of steal steps are bounded by $O(PT_\infty)$ (shown in Theorem 9 in [16]). Naturally, the smaller the probability of stealing into the critical node, the larger the constant hidden in the $O(PT_\infty)$ term, and thus NUMA-WS has a larger constant hidden in front of the T_∞ term.

It is important that a thief flips a coin to decide whether it should steal into a victim's mailbox versus its deque instead of always looking into the mailbox first. The coin flip guarantees that the critical node is stolen with probability at least $1/(2cP)$. If the thief always looks into the mailbox, the critical strand could be at the head of some deque and never gets stolen. Having a mailbox of size one also guarantees that, since there can be only one entry in the mailbox at any given point. A constant-sized mailbox can work, but would complicate the argument and require imposing an ordering on how the mailbox is accessed — a mailbox with multiple entries would need to maintain the top-heavy deques property as well.

Now we consider the extra overhead incurred by work pushing. Since pushing work does not directly contribute to advancing the computation, we need to bound the time steps workers spent pushing work. We will bound the cost of pushing by amortizing it against successful steals and show that only a constant number of pushes can occur per successful steal.

A worker performs work pushing only on full frames that belong to a different socket under these scenarios: a successful non-trivial sync, last spawned child returning to a suspended parent, and a successful steal. For each of such events, one can attribute the event to some successful steal occurred that led to the event. Moreover, there can be at most two such events counted towards a given successful steal, since a frame only performs a nontrivial sync if it has been stolen since its last successful sync. Since only at most two events can be counted towards a successful steal, and only at most constant number of pushes can occur per event due to the pushing threshold (defined in Section III), we can amortize the cost of pushing against successful steals and upper bound that by $O(PT_\infty)$ as well. This amortization argument utilizes the fact that we have a constant pushing threshold and a single-entry mailbox.

Finally, in classic work stealing, the number of additional cache misses on private caches due to parallel execution is simply bounded by the number of successful steals [18] — $O(PT_\infty C)$ for a private cache of size C. The intuition is that, each successful steal forces the worker to refill its private cache. In NUMA-WS, the same bound follows since both the steals and work pushing are bounded by $O(PT_\infty)$.

V. Empirical Evaluation

This section empirically evaluates NUMA-WS. NUMA-WS has similar work efficiency and low scheduling overhead as in Cilk Plus, but it mitigates work inflation and thus provides better scalability. Moreover, NUMA-WS maintains the same processor-oblivious model, and all benchmarks tested scale as the number of cores used increases.

Experimental setup. We ran all our experiments on a 32-core machine with 2.20-GHz cores on four sockets (Intel Xeon E5-4620) with the same configuration shown in Figure 1. 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. All benchmarks are compiled with Tapir [35], a LLVM/Clang based Cilk Plus compiler, with `-O3` running on Linux kernel version 3.10 with NUMA support enabled. Each data point is the average of 10 runs with standard deviation less than 5%. When running the vanilla Cilk Plus, we tried both the first-touch and interleave NUMA policies for each benchmark and used the configuration that led to the best results.

Benchmarks. Benchmark `cg` implements conjugate gradient that solves system of linear equations in the form of $Ax = b$ with a sparse input matrix A. Benchmark `cilksort` performs parallel mergesort with parallel merge. Benchmark `heat` implements the Jacobi-style heat diffusion on a 2D plane over a series of time steps. Benchmark `hull` implements quickhull to compute convex hull. The algorithm works by repeatedly dividing up the space, drawing maximum triangles, and eliminating points inside the triangles. When there are no more points outside of the triangles, we have found the convex hull. Since `hull`'s work and span can differ greatly depending on the input data points, we ran it on two different data sets: one with randomly generated points that lie within a sphere (`hull1`), and another with randomly generated points that lie on a sphere (`hull2`). There is a lot more computation in `hull2`, since the algorithm has a harder time eliminating points. Benchmark `matmul` implements a eight-way divide-and-conquer matrix multiplication with no temporary matrices. Benchmark `strassen` implements a matrix multiplication algorithm that performs seven recursive matrix multiplications and a bunch of additions. Most benchmarks are originally released with MIT Cilk-5 [5], except for `cg`, which comes from the NAS parallel benchmarks [36] and `hull`, which comes from the problem-based benchmark suite [37]. Two benchmarks benefit from the data layout transformation (Section III-C): `matmul` and `strassen`, and we also ran the versions with the data layout transformation (`matmul-z` and `strassen-z`) on both platforms. For a given application, we used the same input and base case sizes for both platforms.

A. Work Efficiency and Scalability

We first provide the overview of our results. Figure 7 shows the T_S, T_1, and T_{32} executing on Cilk Plus and on NUMA-WS. As expected, the T_1 for all benchmarks are similar for the two platforms, since the code are similar with the exception of linking with different scheduler. We compute the spawn overhead by dividing T_1 with the corresponding T_S (shown in parentheses under T_1). Like in Cilk Plus, with appropriate coarsening, NUMA-WS retains high work efficiency. NUMA-WS does not incur any additional overhead on the work term to achieve NUMA awareness. For T_{32}, NUMA-WS was able to achieve better scalability compared to Cilk Plus for most benchmarks (shown in parentheses under T_{32}).

Both `matmul` and `strassen` benefit from the data layout transformation (i.e., comparing with the `-z` version). The

	input size /	*serial*	*Cilk Plus*		*NUMA-WS*	
benchmark	*base case size*	T_S	T_1	T_{32}	T_1	T_{32}
`cg`	$75k \times 75/n/a$	360.00	385.41 (1.07×)	29.39 (13.11×)	384.48 (1.07×)	14.89 (25.82×)
`cilksort`	$1.3e8/1k$	20.38	20.47 (1.00×)	0.96 (21.28×)	20.95 (1.03×)	0.79 (26.58×)
`heat`	$16k \times 16k \times 100/16k \times 10$	83.48	83.05 (0.99×)	13.78 (6.03×)	83.05 (0.99×)	5.95 (13.97×)
`hull1`	$100000k/10k$	4.08	4.12 (1.01×)	0.53 (7.71×)	4.11 (1.01×)	0.45 (9.04×)
`hull2`	$100000k/10k$	44.22	44.95 (1.02×)	3.29 (13.67×)	44.69 (1.01×)	2.09 (21.34×)
`matmul`	$4k \times 4k/32 \times 32$	190.86	191.03 (1.00×)	6.45 (29.60×)	190.39 (1.00×)	6.40 (29.76×)
`matmul-z`	$4k \times 4k/32 \times 32$	73.63	73.64 (1.00×)	2.34 (31.44×)	73.65 (1.00×)	2.35 (31.29×)
`strassen`	$8k \times 8k/16 \times 16$	112.82	111.78 (0.99×)	5.08 (22.00×)	111.99 (0.99×)	5.01 (22.37×)
`strassen-z`	$8k \times 8k/16 \times 16$	80.43	82.03 (1.02×)	3.46 (23.69×)	81.78 (1.02×)	3.47 (23.59×)

Fig. 7. The execution times in seconds for the benchmarks: its serial elision, running on Cilk Plus, and on NUMA-WS. The data layout transformation is applied to two benchmarks: `matmul` and `strassen`, denoted as `matmul-z` and `strassen-z`. The numbers in parentheses under the T_1 columns indicate spawn overhead, (i.e., T_1/T_S). The numbers in parentheses under the T_{32} columns show scalability (i.e., T_1/T_{32}).

blocked Z layout helps when used in matrix multiplications, because the index calculation incurs little additional overhead, and it traverses the matrices in a way that enables the prefetcher.

Beyond data layout transformation, NUMA-WS does not provide more benefit, which is expected, because `matmul` readily obtains good scalability to begin with, and we didn't use locality hints in `strassen`. It's challenging to specify sensible locality hints in `strassen` due to how the algorithm works. Sub-matrices of the inputs are used in different parts of the computation, and thus the data necessarily has to be accessed by multiple sockets. One easy way to specify locality hint for `strassen` is to perform a eight-way divide-and-conquer matrix multiplication at the top-level, and only perform the seven-way divide starting from the second level of recursion. Doing so would allow one to specify locality hint at the top level. We attempted that strategy, but it turns out that, the T_{32} performance of the top-eight-way version is comparable to the version reported with no locality hint. This is because the top-eight-way version indeed have less work inflation, but at the expense of 15% increases in overall T_1, because we are not getting the $O(n^{\lg 7})$ work at the top level. Thus, all things considered, we chose this version over the top-eight-way, since it is more work efficient, and provides comparable T_{32} execution time.

B. Scheduling Overhead and Work Inflation

We examine the detailed breakdown of T_{32} next. Figure 8 shows the work time running on one core, and the work, scheduling, and idle times running on 32 cores for both platforms. We note that, like in Cilk Plus, NUMA-WS has little scheduling overhead. Since the scheduling overhead is already low in Cilk Plus to begin with, the improved scalability of NUMA-WS largely comes from mitigated work inflation.

By comparing W_{32} (work time on 32 cores) with its respective T_1, one can gauge the work inflation for both platforms. Note that one cannot entirely avoid work inflation, since any shared data (i.e., multiple workers taking turns to write to the same memory location) or any work migration (i.e., successful steals) will inevitably incur work inflation. However, compared to Cilk Plus, NUMA-WS indeed mitigates work inflation.

In general, `cg`, `cilksort`, `heat`, and `hull` (both inputs) obtain visible decrease in work inflation when running on NUMA-WS; `matmul` has little to begin with, and `strassen` did not use locality hints as explained in earlier. Benchmark `hull1` has a higher work inflation than `hull2`, because for the particular input used for `hull1`, points were eliminated quickly. Thus, the majority of the computation time is spent doing parallel prefix sum, where the forward and backward propagations touch different parts of the data array and simply does not have much locality.

C. Maintaining Processor-Oblivious Model

Next, we show that, NUMA-WS retains the processor-oblivious programming model and the same benchmarks work well across different number of cores. Figure 9 shows the scalability (T_1/T_P) plot across benchmarks. The workers are packed tightly using the smallest number of sockets. For a given benchmark, its data points are collected using the same user applications without modification. The only thing that changed is the input argument to the runtime specifying the number of cores and sockets to use. Even though we ran the program on different number of sockets, there is no need to change the program. The program queries the runtime how many physical sockets are used during the initialization phase and initializes variables storing the IDs of virtual places accordingly depending on how many sockets are used.

As can be seen in Figure 9 the scalability curves are smooth, indicating that the application indeed gains speedup steadily as we increase the number of cores. An exception is `hull1`, which as explained earlier, spends majority of the computation time on prefix sum, which does not have much locality. Its scalability clearly degrades once we move from a single socket to multiple sockets.

VI. Related Work

Researchers have observed the problems of scaling task parallel programs due to work inflation, and proposed various mechanisms and scheduling policies to mitigate the effect [19]–[29]. None of the proposed platforms achieve all the desired goals, however. In particular, none of them focuses on achieving work efficiency nor provides provably efficient time bound.

One common approach is to utilize some kind of work-stealing hierarchy, where the scheduler employs a centralized shared queue among workers within a socket so as to load balance across sockets. Work described by [20], [22]–[24], [27], [28], [38] take such an approach. In order to aid load balancing across sockets, work described by [20], [23], [38] also propose heuristics to perform the initial partitioning of the work so that

	Cilk Plus				*NUMA-WS*			
benchmark	T_1	W_{32}	S_{32}	I_{32}	T_1	W_{32}	S_{32}	I_{32}
`cg`	385.41	898.60 (2.33×)	10.44	21.48	384.48	443.63 (1.21×)	6.75	17.36
`cilksort`	20.47	31.56 (1.54×)	0.33	0.14	20.95	25.39 (1.21×)	0.15	0.06
`heat`	83.05	435.11 (5.24×)	1.09	2.96	83.05	186.83 (2.25×)	0.43	1.89
`hull1`	4.12	16.71 (4.05×)	0.08	0.09	4.11	14.50 (3.53×)	0.05	0.14
`hull2`	44.95	102.45 (2.28×)	0.17	0.23	44.69	69.62 (1.56×)	0.16	0.28
`matmul`	191.03	207.78 (1.09×)	0.25	0.15	190.39	202.79 (1.07×)	0.40	0.52
`matmul-z`	73.64	74.99 (1.02×)	0.25	0.11	73.65	74.79 (1.02×)	0.28	0.29
`strassen`	111.78	168.17 (1.50×)	0.39	0.52	111.99	168.28 (1.50×)	1.22	0.39
`strassen-z`	82.03	119.44 (1.46×)	0.88	0.75	81.78	118.34 (1.45×)	2.29	0.43

Fig. 8. T_1 shows the one-core running time on each platform. W_{32}, S_{32}, and I_{32} show the work time, scheduling time, and idle time, respectively, when running on 32 cores. The numbers in parentheses next to W_{32} indicates the work inflation (i.e., W_{32}/T_1) compared with the T_1 from the same platform.

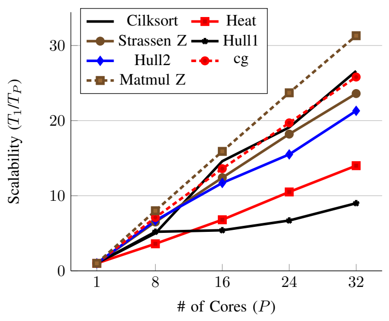

Fig. 9. The scalability of benchmarks running on NUMA-WS. The x-axis shows P, the number of cores used, and the y-axis shows T_1/T_P, the scalability. Threads are packed onto sockets tightly and the smallest number of sockets is used, i.e., for 24 cores, 3 sockets are used.

work distributed to different sockets can be somewhat balanced to begin with. It's not clear the scheduler proposed by these prior work provide the same theoretical guarantees as the classic work stealing. Moreover, if the initial partitioning was not done in a load-balanced way, the performance may degrade, since load balancing among sockets require more centralized control. In contrast, our work still utilizes randomized work stealing and provides sound guarantees.

Besides having a hierarchy for work stealing, prior work [27], [28] also explored the notion of work pushing, but the key distinction between these work and ours is that, NUMA-WS performs work pushing in a work-efficient way. Without judicious selection of when to performing work pushing, the scheduler can incur high pushing overhead on the work term, causing the parallel overhead (T_1/T_S) to increase.

Related to work-pushing, the notion of mailbox is first proposed by [18]. In their paper, Acar et. al provided the analysis for how to bound cache misses for private caches for the classic work stealing algorithm. The paper also proposed a heuristic for obtaining better locality for iterative data parallel code, where the program iteratively executes a sequence of data-parallel loops that access the same set of data over and over. The assumption is that, the program can obtain better locality if the runtime can keep the same set of data-parallel iterations on the same worker. Upon a spawn, if the work item being pushed onto the deque has a different affinity (as determined by the data-parallel loop indices) from the executing worker, the work item is pushed onto both the executing worker's deque and the designated worker's mailbox, a FIFO queue with multiple entries. The use of mailbox is proposed as a heuristic, and their analysis does not extend to include the heuristic.

The use of mailbox is subsequently incorporated into Intel Threading Building Blocks (TBB) [2], [3]. Majo and Gross extended TBB to be NUMA-aware in a portable fashion. Our proposed programming API took inspiration from them. However, their work has similar downside that, it's not work efficient and does not provide provable guarantee.

Work by [19] also utilizes the notion of places, and the programmer can specify that a spawned task being executed at a specific place. The scheduler restricts such tasks to be executed exactly at the designated place, however, which can impede scheduler's ability to load balance and thus leads to inefficient execution time bound.

Work by [39] targets specifically workload that performs the same parallel computation over and over and thus their mechanism is designed specifically for the set of programs that exhibit such behaviors. In their work, the workers records the steal pattern during the first iteration, and replay the same steal pattern in subsequent iterations, thereby gaining locality.

Work by [26], [40] studies a space-bounded scheduler, which provides provable guarantees for bounding cache misses for shared caches, but may sacrifice load balance as a result.

Work by [41] proposes a locality-ware task graph scheduling framework, which provides a provably good execution time bound. However, the framework is designed for task graph computations which has a different programming model.

Finally, work by [42] proposed a locality-aware scheduler called ***localized work stealing***. Give a computation with well-defined affinities for work items, each worker keeps a list of other workers who might be working on items belonging to it. That is, whenever a thief steals a work item, it will check the affinity of the work item and add its name onto the owner's list (who might be a different worker from the victim). Whenever a worker runs out of work, it checks the list first and randomly select a worker from the list to steal work back. This steal-back mechanism, like the lazy work pushing in our work, can be amortized against steals. However, since a worker is required to check the list when it runs out of work to do their bound is slightly worse. The work is primarily theoretical and has not been implemented.

VII. Conclusion and Future Direction

In this paper, we have shown that NUMA-WS, like the classic work stealing, provides strong theoretical guarantees. Moreover, its implementation is work efficient and can provide better scalability by mitigating work inflation. We conclude by discussing some of its limitations and potential future directions. NUMA-WS is designed to give the programmer a finer control over where a task is executed. In order to mitigate work inflation, however, the programmer still needs to provide locality hints and allocates / partitions data in such a way that allows the task and its data to be co-located. First, the programmer needs to use the runtime to query the number of sockets and perform the appropriate data partitioning, and thus cannot be entirely socket oblivious. Second, it may be challenging to partition data and provide sensible locality hints for algorithms that perform random memory accesses (i.e., a task may access data scattered across sockets). Examples of such algorithms include various graph algorithms, or `strassen` we tested. While NUMA-WS won't degrade the performance of such algorithms, it won't bring benefit, either. Interesting future directions include devising a programming interface that allows the programmer to be socket oblivious and investigating how one may mitigate NUMA effects in algorithms where it's challenging to co-locate a task and its data.

ACKNOWLEDGMENTS

We thank Suyeon Kang and Ramsay Shuck for their contributions to prior iteration of the system. We thank the reviewers for the helpful feedback on the earlier revisions of the paper. Finally, we thank Kelly Shaw for her tireless shepherding and thoughtful suggestions on how to improve the paper.

REFERENCES

[1] *OpenMP Application Program Interface, Version 4.0*, Jul. 2013.

[2] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[3] *Intel(R) Threading Building Blocks*, Intel Corporation, 2012, available from http://www.threadingbuildingblocks.org/documentation.php.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Jornal of Parallel and Distributed Computing*, vol. 37, pp. 55–69, 1996.

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

[6] 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.

[7] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson, "Using memory mapping to support cactus stacks in work-stealing runtime systems," in *PACT*. ACM, 2010, pp. 411–420.

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

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

[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*, 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] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *FOCS*, November 1994, pp. 356–368.

[15] ——, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[16] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *SPAA*, 1998, pp. 119–129.

[17] ——, "Thread scheduling for multiprogrammed multiprocessors," *Theory of Computing Systems*, pp. 115–144, 2001.

[18] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *SPAA*, 2000, pp. 1–12.

[19] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems," in *IPDPS*, 2010, pp. 1–12.

[20] Q. Chen, Z. Huang, M. Guo, and J. Zhou, "CAB: Cache aware bi-tier task-stealing in multi-socket multi-core architecture," in *ICPP*, 2011, pp. 722–732.

[21] 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.

[22] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore numa systems," *International Journal of High Performance Computing Applications*, vol. 26, no. 2, pp. 110–124, May 2012.

[23] Q. Chen, M. Guo, and Z. Huang, "CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures," in *ICS*, 2012, pp. 163–172.

[24] S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins, "Characterizing and mitigating work time inflation in task parallel programs," in *SC*, 2012, pp. 65:1–65:12.

[25] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, "Locality-aware task scheduling and data distribution on numa systems," in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer Berlin Heidelberg, 2013, pp. 156–170.

[26] H. V. Simhadri, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and A. Kyrola, "Experimental analysis of space-bounded schedulers," in *SPAA*, 2014, pp. 30–41.

[27] A. Drebes, K. Heydemann, N. Drach, A. Pop, and A. Cohen, "Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 3, pp. 30:1–30:25, Aug. 2014.

[28] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach, "Scalable task parallelism for numa: A uniform abstraction for coordinated scheduling and memory management," in *PACT*, 2016, pp. 125–137.

[29] Z. Majo and T. R. Gross, "A library for portable and composable data locality optimizations for numa systems," *ACM Transactions on Parallel Computing*, vol. 3, no. 4, pp. 20:1–20:32, March 2017.

[30] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM*, vol. 21, no. 2, pp. 201–206, Apr. 1974.

[31] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, Mar. 1969.

[32] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other Cilk++ hyperobjects," in *SPAA*, 2009, pp. 79–90.

[33] U. A. Acar, A. Charguéraud, and M. Rainey, "Parallel work inflation, memory effects, and their empirical analysis," arXiv:1709.03767 [cs.DC], 2017.

[34] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *FOCS*, Oct. 1999, pp. 285–297.

[35] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into LLVM's intermediate representation," in *PPoPP*, 2017, pp. 249–265.

[36] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS parallel benchmarks—summary and preliminary results," in *ACM/IEEE Supercomputing*, 1991.

[37] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: the Problem Based Benchmark Suite," in *SPAA*, 2012.

[38] Q. Chen, M. Guo, and H. Guan, "LAWS: Locality-aware work-stealing for multi-socket multi-core architectures," in *ICS*, 2014, pp. 3–12.

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

[40] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri, "Scheduling irregular parallel computations on hierarchical caches," in *SPAA*, 2011, pp. 355–366.

[41] J. Maglalang, S. Krishnamoorthy, and K. Agrawal, "Locality-aware dynamic task graph scheduling," in *ICPP*, Aug 2017, pp. 70–80.

[42] W. Suksompong, C. E. Leiserson, and T. B. Schardl, "On the efficiency of localized work stealing," arXiv:1804.04773 [cs.DC], 2018.