PufferFish: NUMA-Aware Work-stealing Library using Elastic Tasks

Vivek Kumar
IIIT-Delhi, New Delhi, India

***Abstract*—Due to the challenges in providing adequate memory access to many cores on a single processor, Multi-Die and Multi-Socket based multicore systems are becoming mainstream. These systems offer cache-coherent Non-Uniform Memory Access (NUMA) across several memory banks and cache hierarchy to increase memory capacity and bandwidth. Random work-stealing is a widely used technique for dynamic load balancing of tasks on multicore processors. However, it scales poorly on such NUMA systems for memory-bound applications due to cache misses and remote memory access latency. Hierarchical Place Tree (HPT) [1] is a popular approach for improving the locality of a task-based parallel programming model, albeit it requires the programmer to map the dynamically unfolding tasks over a NUMA system evenly. Specifying data-affinity hints provides a more natural way to map the tasks than HPT. Still, a scalable work-stealing implementation for the same is mostly unexplored for modern NUMA systems.**

This paper presents *`PufferFish`*, a new `async–finish` parallel programming model and work-stealing runtime for NUMA systems that provide a close coupling of the data-affinity hints provided for an asynchronous task with the HPTs in Habanero C/C++ library (`HClib`). `PufferFish` introduces *Hierarchical Elastic Tasks (HET)* that improves the locality by shrinking itself to run on a single worker inside a `place` or puffing up across multiple workers depending on the work imbalance at a particular `place` in an HPT. We use a set of widely used memory-bound benchmarks exhibiting regular and irregular execution graphs for evaluating `PufferFish`. On these benchmarks, we show that `PufferFish` achieves a geometric mean speedup of 1.5× and 1.9× over HPT implementation in `HClib` and random work-stealing in `CilkPlus`, respectively, on a 32-core NUMA AMD EPYC processor.

***Index Terms*—NUMA; async-finish Programming Model; Work-Stealing;**

I. Introduction

The breakdown of Dennard scaling has put a stop on faster single-core performance, and mainstream processors today are using multicores for achieving better performance. However, modern CPUs are way faster than DRAM. Hence, it is difficult for the processor vendors to maintain low memory latency and high memory bandwidth by stamping many cores onto a single processor. Complex hardware caching mechanisms and on-chip memory controllers can alleviate the situation up to some level, but it also significantly increases the processor manufacturing cost. To get around this problem, modern processors support cross-chip interconnect. It helps bridge several multicore dies and processors (sockets) together in a cache coherent manner, where each unit has its local DRAM and caches. Still, it can also access the memory on the remote units. This architecture is becoming mainstream and is called cache-coherent Non-Uniform Memory Access (NUMA) architecture due to different layers of the memory hierarchy. Figure 1 represents the NUMA architecture of the AMD EPYC 7551 processor used in this paper for experimental evaluations.

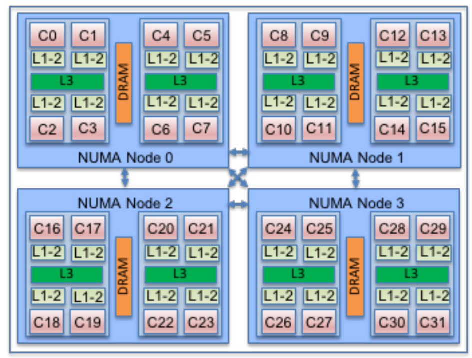


Fig. 1. AMD EPYC 7551 processor

Tasks-based programming models [2], [3], [4], [5], [6] are widely used for writing parallel programs for multicore processors. They rely on an underlying work-stealing [7] runtime for dynamic load-balancing of the tasks exposed by the programmer. Work-stealing uses a pool of worker threads where each worker maintains a local set of tasks (*deque*). Once a worker (*thief*) becomes idle, it *randomly* chooses a busy worker (*victim*) to *steal* a task. Random victim selection has been analyzed and shown to achieve a provably good parallel speedup [7]. However, randomly selecting a victim on NUMA systems can adversely affect the performance of a memory-bound application due to cache misses and remote memory access latency. In iterative applications, the locality of random work-stealing can be improved by using runtime profiling either by using trace-replay based constrained work-stealing [8] or by profiling hardware performance counters [9]. However, the trace-replay of random work-stealing does not guarantee to co-locate the task and its data on the same NUMA domain. Likewise, the hardware performance counter-based approach suffers from portability issues as these counters are specific to a particular processor architecture.

Hierarchical Place Tree (HPT) is another popular approach for improving the locality based on programmer's insights on mapping tasks onto a hierarchical representation of places on NUMA systems [1], [10], [11]. It has been widely adopted across several task-based programming models [12], [13], [14], where each implementation uses some form of hierarchical work-stealing where a thief first chooses a nearby victim in the NUMA memory hierarchy before attempting a remote steal. A limitation of this technique is that it can work well only if the work-load is evenly partitioned across all the place in the NUMA system. Otherwise, it can cause starvation. The programmer's insights on mapping tasks can work well for applications with regular execution Directed Acyclic Graph (DAG). Still, it is challenging to determine an optimal partitioning a priori for applications with irregular execution DAG. If a DAG has the *same* branching degree for all its non-leaf nodes, then it is a regular DAG and an irregular DAG in *different* branching degrees.

This paper explores a simple and straightforward approach for NUMA-aware work-stealing called PufferFish that does not use any runtime profiler and improves the locality of both regular and irregular DAGs. PufferFish extends the traditional **async–finish** programming in the HClib library [4] by allowing the programmer to provide data-affinity hints [15] in an **async** instead of mapping a task to a place in an HPT. PufferFish uses a novel hierarchical work-stealing algorithm that parses these hints and dynamically decides the task's optimal execution location. This location could either be a particular worker, a group of workers inside a NUMA domain, or a locality-free location. For improving the locality of tasks within a place, it introduces a hierarchical implementation of elastic tasks [16], called Hierarchical Elastic Task (HET). HET operates only at leaf place in an HPT that corresponds to shared caches (e.g., L3). Once a worker pulls a task into a leaf place, that task shrinks itself into a sequential task such that it can run only on that particular worker. Whenever load-imbalance arises at a leaf place, HET running on other workers in that same leaf place expands itself into parallel tasks for balancing that specific place's workload. We choose a set of widely-used micro-benchmarks and a real-world application and implement regular and irregular DAG versions to study the performance benefits of PufferFish on a modern NUMA processor. We show that PufferFish performs significantly better than the random work-stealing implementations in HClib (Section II-A) and CilkPlus [6].

In summary, this paper makes the following contributions:

- *PufferFish*, a new **async–finish** task-based parallel programming model for NUMA systems that integrates data-affinity hints with an HPT implementation.
- A novel hierarchical work-stealing runtime for PufferFish with the support for *Hierarchical Elastic Tasks* that improves the locality by shrinking itself to run on a single worker inside a place, or by puffing up across multiple workers depending on the work imbalance at a particular place in an HPT.
- Performance evaluation of PufferFish compared to HClib and CilkPlus on a 32-core NUMA AMD EPYC 7551 processor using regular and irregular execution DAGs in four popular memory-bound applications.

The rest of the paper is structured as follows. Section II provides the relevant background. Section III motivates and explain the PufferFish programming model. Section IV explains the design and implementation of PufferFish work-stealing runtime. Section V discusses our evaluation methodology. Section VI discusses the performance evaluation of PufferFish. Section VII explains the related work and finally section VIII concludes the paper.

```
void foo(int n) {
  int a,b;
  hclib::finish([&a,&b,n]() { /*start finish scope*/
    hclib::async([&a,n]() {
      a = S1(n);
    });
    b = S2(n);
  }); /*end finish scope*/
  S3(a,b);
}
```

Fig. 2. An example of **async–finish** programming using HClib. **async** denotes a task that could run in parallel to other tasks and **finish** denotes synchronization point for parallel tasks created within its scope.

II. Background

This section provides a brief overview of the **async–finish** parallel programming model supported by Habanero C/C++ library and its work-stealing runtime (Section II-A) and the Hierarchical Place Trees implementation in HClib (Section II-B).

A. *HClib Library*

The Habanero-C/C++ library (HClib) [4] offers an **async–finish** programming model for exploiting shared memory parallelism. These constructs were first coined by X10 language [17]. Now it has been adopted by several other frameworks supporting task parallelism [18], [19]. Its variants are also supported in other popular frameworks [20], [2].

a) ***async–finish*** *programming model:* HClib is a native library-based implementation of the Habanero programming model that offers C and C++ APIs. It provides high productivity in writing **async–finish** programs by making heavy use of C++11 lambda functions in its APIs. C++11 lambdas avoid the need for compiler support while still retaining the syntactic convenience of language-based approaches. Figure 2 shows a sample code written by using **async–finish** APIs supported by HClib. Both these APIs accept a C++11 lambda function as an argument. The **async** API creates a task S1, which can run in parallel with the following statements, i.e., S2. An **async** is a powerful primitive because it can be used to enable any statement to execute as a parallel task, including statement blocks, **for**-loop iterations, and function calls. **finish** is a generalized join operation. Method S3 will never execute until both S1 and S2 have completed. The power of these constructs comes from the ability to nest async and finish arbitrarily. Due to its simple programming interface, HClib is also used for teaching an introductory parallel programming course at IIIT-Delhi (CSE502).

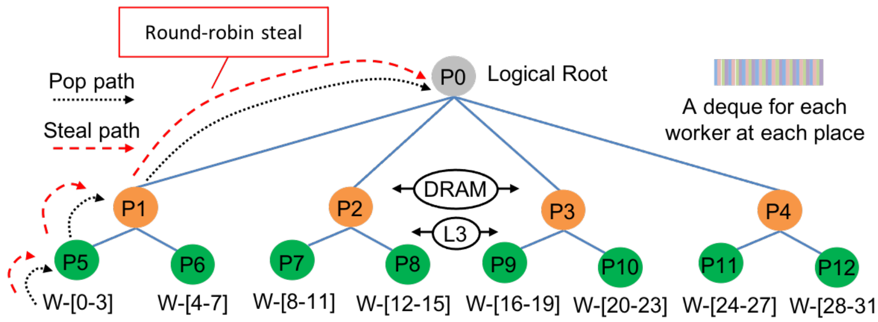


Fig. 3. Hierarchical work-stealing implementation in `HClib` for HPT shown in Figure 4(a)

```
<HPT version="1.0">
  <place num="1" type="mem"> <!-- logical root -->
    <place num="4" type="cache"> <!-- 4 NUMA nodes -->
      <place num="2" type="cache"> <!-- 2 L3 per node -->
        <worker num="4"/> <!-- 4 workers per L3 -->
      </place>
    </place>
  </place>
</HPT>
```

(a) XML file for representing the hierarhcy shown in Figure 3

```
<HPT version="1.0">
  <place num="1" type="mem"> <!-- logical root -->
    <place num="4" type="cache"> <!-- 4 NUMA nodes -->
      <worker num="8"/> <!-- 8 workers per node -->
    </place>
  </place>
</HPT>
```

(b) XML file for the same machine but without adding L3 cache

Fig. 4. User provided XML file for representing the hierarhcy shown in Figure 3.

b) Help-first work-stealing runtime: `HClib` internally uses a *help-first* [21] work-stealing implementation for load-balancing of parallel tasks compared to *work-first* [20] work-stealing approach in Cilk due to library-based implementation. In this help-first implementation, the worker (`victim`) executing the **`async`** task `S1` in Figure 2 would `push` this task on its `deque`. It will then execute the statement `S2`. After completing the execution of `S2`, the worker will reach the end **`finish`** scope, where it will try to `pop` the task `S1` from its `deque`. If any thief stole this task, the victim would become a thief. Otherwise, it will `pop` and execute `S2` after a context switch. In work-first technique, a context-switch is triggered only by a `steal` operation.

B. Hierarchical Place Trees (HPT)

Languages such as X10 [22] and Chapel [23] introduced `place` and `locale`, respectively, to specify the task locality in a NUMA cluster. However, this model is flat in structure and cannot capture the locality as per the NUMA hierarchy. Habanero programming model generalized this notion of `place` as an abstraction of the memory hierarchy in a NUMA system [1], [10]. This model is also being used in other task-based programming models [12], [13], [11], [4].

`HClib` allows the user to define the NUMA memory hierarchy using an XML input file. Figure 4 shows the code for two XML files for representing the memory hierarchy of the AMD processor shown in Figure 1 in two different ways. In this paper we have used the implementation shown in Figure 4(a). If an XML file is not provided, `HClib` assumes a flat memory hierarchy (single `place`). Runtime will read this XML file at the program launch and model the affinity as a tree of places, thus the name Hierarchical Place Trees (HPT). The choice of a particular representation of hierarchy often depend on the application, and the desired trade-off between locality and load balance for a given task. Allowing the programmer to specify this configuration in a flexible way is an added advantage.

Each `place` in an HPT will contain a `deque` for each worker for lock-free `push` and `pop` operations. Although synchronization is required for `steal`. Any worker can push a task at any `place` using an API "`async_at(place_id, lambda)`". This has a downside that the programmer is required to specify the `place` hint with each `async_at` tasks. This is challenging for recursive parallel programs. For preserving the locality, HPT restricts the `pop` and `steal` at a `place` only to the workers in the child nodes of this `place` in HPT. For example, in Figure 3, worker `W0` will first `pop` from `place P5`, followed by `place P1` and `place P0`, respectively. If it failed in `pop`, it would attempt to `steal` from these same places in the same order. This helps in achieving locality among the tasks that share some data. `HClib` runtime ensures that the worker threads are bounded to the appropriate core id for avoiding thread migration.

III. PufferFish Programming Model

We had to ensure that `PufferFish` preserves the serial-elision [20] as much as possible. The basic **`async–finish`** programming model in `HClib` supports this property that is removing all parallel APIs results in a valid sequential program. It is otherwise hard to achieve by using a programmer's insights on mapping tasks on a NUMA system. To achieve this goal, `PufferFish` extends the **`async–finish`** programming supported by `HClib` with two sets of new APIs:

- *NUMA-aware memory allocations*: The programmer should first allocate the arrays accessed inside a parallel region by using

```
1 #include <hclib.hpp>
2 using namespace pufferfish;
3 int* A;
4 /* recursive merge generates two async_hinted */
5 void merge(int L1, int H1, int L2, int H2);
6 void sort(int L, int N) {
7   if(N<LIMIT) return seq_sort(L, N);
8   int Q=N/4;
9   finish([=]() {
10    async_hinted(A, L, L+Q-1, [=]() {
11      sort(L, Q);
12    });
13    async_hinted(A, L+Q, L+2*Q-1, [=]() {
14      sort(L+Q, Q);
15    });
16    async_hinted(A, L+2*Q, L+3*Q-1, [=]() {
17      sort(L+2*Q, Q);
18    });
19    async_hinted(A, L+3*Q, L+N-1, [=]() {
20      sort(L+3*Q, Q);
21    });
22  });
23
24  finish([=]() {
25    async_hinted(A, L, L+2*Q-1, [=]() {
26      merge(L, L+Q-1, L+Q, L+2*Q-1);
27    });
28    async_hinted(A, L+2*Q, L+N-1, [=]() {
29      merge(L+2*Q, L+3*Q-1, L+3*Q, L+N-1);
30    });
31  });
32
33  merge(L, L+2*Q-1, L+2*Q, L+N-1);
34 }
35 int main(int argc, char** argv) {
36  launch([&]() {
37    A = numa_alloc_blockcyclic<int>(N);
38    initialize(A);
39    sort(0, N);
40    numa_free(A);
41  });
42 }
```

Fig. 5. Recursive CilkSort benchmark parallelized using `PufferFish` programming model. Underlined APIs are specific to `PufferFish`

numa_alloc_blockcyclic<T>(count) and **numa_alloc_interleave**<T>(count) APIs. In this prototype implementation of `PufferFish` we currently support one-dimensional arrays only. **numa_alloc_blockcyclic** block-cyclic performs distribution of the physical pages over NUMA nodes with block size as `num_pages/num_numa_nodes`. **numa_alloc_interleave** performs a round-robin distribution of physical pages across all NUMA nodes. **numa_free** API is used for freeing the memory. All these APIs are wrappers over `libnuma` library [24].

- *Providing data-affinity hints*: An **async_hinted** API is used for this purpose that is otherwise a regular **async**. It is a variadic function that accepts a variable number of affinity hints. Each hint is a pair of three variables in the following order: pointer to the start index of the array allocated using **numa_alloc_blockcyclic** and **numa_alloc_interleave** APIs, `start` and the `end` indices in this array touched by this task. The last parameter to **async_hinted** is the lambda task. The programmer should judiciously pass the data-affinity hints for achieving good performance. If an **async_hinted** operates on different arrays allocated using the same API, of the same datatype and on the same index range, then hint should be provided only for one of the variety.

To further motivate `PufferFish`, we show its usage in Figure 5 using a recursive CilkSort program used in our experimental evaluations (Section V). This program generates an irregular execution DAG as each vertex's degree in the DAG is not the same. To understand programmer-based partitioning challenges, consider two different NUMA systems, namely *System-A* having two NUMA nodes, and *System-B* having four NUMA nodes. There are three sets of **async–finish** scopes at the top-level that requires partitioning by the programmer: a) *Region-1* inside `sort` method with four parallel tasks (Lines 9– 22), b) *Region-2* inside `sort` method with two parallel tasks (Lines 24– 31) and c) *Region-3* inside the `merge` method with two parallel tasks (hidden in Figure 5). Partitioning the *Region-1* is easy for both the systems. However, it is challenging to partition the *Region-2* and *Region-3* evenly over *System-B* without modifying the above algorithm. Programmer-based top-level partitioning will also break the serial-elision. `PufferFish` overcomes this limitation by relying on data-affinity hints from the programmer instead of top-level task partitioning. Except for the **numa_alloc_blockcyclic** (Line 37) and **numa_free** (Line 40) APIs, it supports the serial-elision property. Removing the lambda function APIs for **finish**, **async_hinted**, and **launch** will recover the sequential implementation of CilkSort.

IV. Design and Implementation

In this section, we describe our implementation of `PufferFish` that is based on `HClib` work-stealing library (Section II-A). Our implementation and the benchmarks are released open source online on GitHub [25].

At a high-level, `PufferFish` implementation first uses the data-affinity hints provided in an **async_hinted** for finding the `place` id in an HPT that contains most of the physical pages for the memory accessed in this task (Section IV-B). The **async_hinted** is then pushed at the current worker's `deque` at that `place`. When a worker become idle, it would attempt to grab a task using hierarchical work-stealing that improves the default implementation in `HClib` by reducing starvation (Section IV-C). For minimizing the loss in locality arising due to context switches at every task execution, `PufferFish` uses a hierarchical elastic task implementation of **async_hinted** that can inflate or deflate its recursive parallelism depending on the `place` load (Section IV-D).

A. NUMA memory manager

`PufferFish` has a NUMA memory manager that is built over `libnuma` library. It currently supports block-cyclic (total NUMA node number of blocks) and interleave allocation of a contiguous chunk of memory. Each allocation stores a pointer to the allocated memory, size of the datatype, allocation length, and the type of distribution. This information is used for mapping an array access range to its physical pages.

```
1  place_t* find_place(hint_info_t* list, int size) {
2    int node_array[num_numa_node()];
3    int multi_node=0, num_pages;
4    for(int i=0; i<size && multi_node<=size/2; i++) {
5      alloc_t type = get_allocType(list[i].ptr);
6      int start_node, end_node;
7      if(type == BLOCK_CYCLIC) {
8        start_node=list[i].start/get_blockSize(list[i].ptr);
9        end_node=list[i].end/get_blockSize(list[i].ptr);
10       num_pages = (end-start)/PAGE_SIZE;
11     } else {  /* type=INTERLEAVE */
12       size_t page_lb=get_pageID(list[i].ptr, list[i].start);
13       size_t page_hb=get_pageID(list[i].ptr, list[i].end);
14       start_node=page_lb % num_numa_node();
15       end_node=page_ub % num_numa_node();
16       num_pages=1;
17     }
18     if(start_node != end_node) multi_node++;
19     else node_array[start_node]+=num_pages;
20   }
21   if(multi_node>size/2) return logical_root_place();
22   place_t* pl=node_with_maxpages(node_array);
23   if(pl==current_worker_numaNode()) return my_leafPlace();
24   else return pl;
25 }
```

Fig. 6. Pseudocode to identify the best `place` to execute an **`async_hinted`**

B. Mapping data-affinity hints to `place` in HPT

Figure 6 shows the pseudocode of the method `find_place` used inside an **`async_hinted`** for finding the optimal `place` for execution, i.e., the `place` containing the maximum number of physical pages of the memory accessed inside an **`async_hinted`**. We first iterate over all the hints provided by the user (Line 4), and for each hint, we calculate the NUMA nodes that contain the physical pages for `start` and `end` indices of the array access (Lines 7– 17). NUMA memory manager APIs `get_blockSize` and `get_pageID` assist in this calculation. This iteration is aborted if half of the hints are for memory ranges spanning over multiple NUMA nodes (Line 18). In this case, the logical root is returned as the optimal `place` (Line 21). For a recursive program, this would frequently happen at the start of recursion. If the iteration completed successfully, the runtime would shortlist the NUMA node containing the maximum number of pages (Line 22). If this node is the home NUMA node of the current worker (Line 23), the optimal place of execution is the worker's leaf place (L3 cache), else the remote DRAM `place` (Line 24).

C. Hierarchical Work-Stealing

Figure 7 shows the Hierarchical Work-Stealing (HWS) in `PufferFish` that is a modification of `HClib`'s implementation of hierarchical `pop` and `steal`. Similar to `HClib`, `PufferFish` also uses total worker count number of `deque` at each `place`. We refer to the HPT shown in Figure 7 for explaining HWS, but `PufferFish` can work with other HPTs as well.

a) `pop` implementation: Unlike `HClib`, a victim in `PufferFish` do not attempt to `pop` from all its parent `place` if it fails to `pop` from its leaf `place`. Recall from Section II-A, the HPT programming model in `HClib` requires the programmer to specify the `place` of execution. A worker can thus create a task in any `place`. This is not true in `PufferFish` as its worker schedules an **`async_hinted`** on HPT based on the data-affinity hints (Section IV-B). For example, worker `W0` shown in Figure 7 can `push` **`async_hinted`** only at `place P0`, `P2`, `P3`, `P4`, and `P5`. It cannot `push` at `P1` (Figure 6, Line 23). A `pop` from `P2-P4` would break the data-affinity hints. The `place P0` contains tasks whose locality is not yet determined. In contrast, `place P1` and `P6` may already contain tasks pushed by other workers guaranteed to have data-affinity to the NUMA node of worker `W0`.

b) `steal` implementation: `PufferFish` does not follow the `steal` path of `HClib` as it can cause load-imbalance. Consider two **`async_hinted`** tasks `T1` and `T2` available at the `place P1` shown in Figure 7. Both `T1` and `T2` are recursive tasks containing two and four **`async_hinted`**, respectively (affinity to `P1`). `T1` is stolen by the thief `W0`, whereas the theif `W4` steals `T2`. As the affinity of child tasks of `T1` and `T2` are at `place P1`, `W0` would `push` two-child tasks of `T1` at `place P5`, whereas `W4` would push four child tasks of `T2` at `place P6`. Clearly, following the `steal` path of `HClib` would create load-imbalance across `P5` and `P6`. Worker `W0` in `PufferFish` avoids this by always attempting its steal from a `place` in the fixed order of `P5`, `P1`, `P6`, and finally from `P0`. `W0` tries to steals from `P1` before `P6` for avoiding the cache misses at `P6`. It attempts to `steal` from `P0` in the end as `P0` only contains the tasks with unresolved affinity. Thieves in `PufferFish` attempt to `steal` from a `place` only after checking a **`boolean`** flag on that `place` that indicates whether this `place` is idle or busy. This is to avoid the overhead of iterating over the O(n) number of `deque` at an idle `place`. The thief who fails to `steal` from a `place` the first time would flip the flag to **`false`**, which would be set to **`true`** only when a victim `push` a task at this `place`.

D. Hierarchical Elastic Tasks

Recall from Section II-A, every task in `HClib` is executed after a context switch. This hampers the locality. To reduce these context switches, **`async_hinted`** in `PufferFish` acts as a Hierarchical Elastic Task (HET) that can shrink or puff up its parallelism depending on the `place` of execution in HPT. If an **`async_hinted`** is executing at the logical root `place P0` in Figure 7 or at NUMA node `place P1`, `P2`, `P3`, and `P4`, it behaves normally without changing its parallelism. When a worker at a leaf `place`, e.g., `W0`, pulls it to `place P5`, it first checks if there is any load imbalance at `place P5`, i.e., if there was any failed steal attempt. If all the workers `W1-W3` are busy, then `W0` will shrink the parallel **`async_hinted`** task into a sequential task for preserving its locality by avoiding context switches. This sequential task is then directly executed by `W0` without further any `push` operations at its deque in `place P5`. It will continue doing this until a failed steal is registered at `place P5`. In that case, `W0` will temporarily resume the parallel execution of

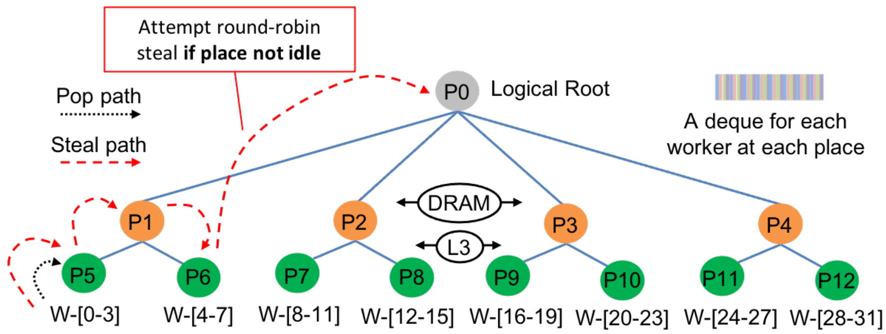


Fig. 7. Hierarchical work-stealing implementation in `PufferFish`

Benchmark	Brief Description	Configuration	Execution DAG	Total occurances of **`async`** API	Total **`async`** tasks created in `HClib`	Source
CilkSort	Mergesort with recursive sorting and merging	8GB of long type array. Threshold=1024	Irregular	8	11M	Cilk
LULESH	Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) proxy application [26]	N=32	Regular	50	12M	HJlib [18]
LULESH-ir		Same as above	Irregular	150	12M	HJlib
SOR	Successive Over-Relaxation (SOR)	double type matrix of size 16K. Threshold=2. Iterations=50	Regular	2	819K	HClib
SOR-ir		Same as above	Irregular	6	821K	HClib
SRAD	Speckle Reducing Anisotropic Diffusion (SRAD)	double type matrix of size16K. Threshold=1. Iterations=50	Regular	4	327K	Rodinia [27]
SRAD-ir		Same as above	Irregular	12	327K	Rodinia

TABLE I
BENCHMARKS USED IN THIS PAPER. AN **`ASYNC`** REPRESENTS **`ASYNC_HINTED`** FOR `PUFFERFISH` AND `CILK_SPAWN` FOR `CILKPLUS`

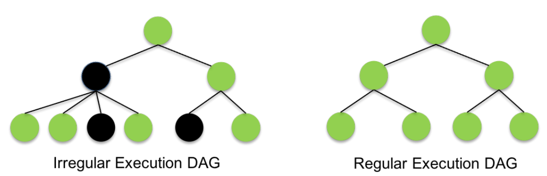


Fig. 8. LULESH-ir, SOR-ir, and SRAD-ir uses the irregular execution DAG, whereas LULESH, SOR, and SRAD uses the regular execution DAG. Non-root and non-leaf green and black nodes have degrees three and five, respectively. Each node is a parallel task.

`async_hinted` by pushing a few tasks to its deque on `place P5`. It will continue this cycle of shrinking and puffing the parallelism in **`async_hinted`** until it becomes a thief again.

V. EXPERIMENTAL METHODOLOGY

Before presenting the evaluation of `PufferFish`, we first describe our experimental methodology.

We have targeted four widely-used benchmarks for our experimental evaluations. We chose these benchmarks as they have been used in several prior works. Table V describe these benchmarks and their respective configurations used in our experimental evaluations. These benchmarks differ in terms of their memory access density and default support for locality. Also, the sequential overheads [19] in these benchmarks are negligible on `HClib`. Further, we created two different implementations of LULESH, SOR, and SRAD benchmarks. Originally these benchmarks contain **`for`**-loop parallelism. We followed the technique described in work by Chen et al. [9], [28], and converted this loop-level parallelism into regular and execution DAG, as shown in Figure 8.

Five different versions of each benchmark were used: a) `HClib` implementation that uses **`async–finish`** APIs and uses random work-stealing, b) `CilkPlus` implementation that uses `cilk_spawn-cilk_sync` APIs and also uses random work-stealing, c) `PufferFish` implementation that uses **`async_hinted-finish`** APIs with data-affinity hints, and uses `PufferFish` implementation of hierarchical work-stealing (Section IV), d) `HClib (HPT_DA)` implementation that uses **`async_hinted-finish`** APIs with data-affinity hints, but use default HPT implementation of `HClib` for hierarchical work-stealing (Section II-B), and e) `Sequential` implementation obtained by using serial-elision. We evaluated `HClib` and `CilkPlus` implementation by using both first-touch policy in Linux and interleave allocation policies supported by `numactl` library. First-touch policy in `HClib` and `CilkPlus` are reported as `HClib (FT)` and `CilkPlus (FT)`, respectively, whereas interleave allocation policy is reported as `HClib (IL)` and `CilkPlus (IL)`, respectively.

The GCC compiler version was 7.5.0. We used the `CilkPlus` version (*libcilkrts5*) shipped with this GCC compiler. `HClib` implementation used in this paper is down-

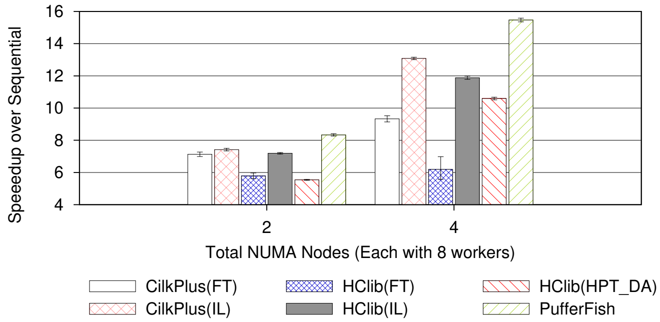
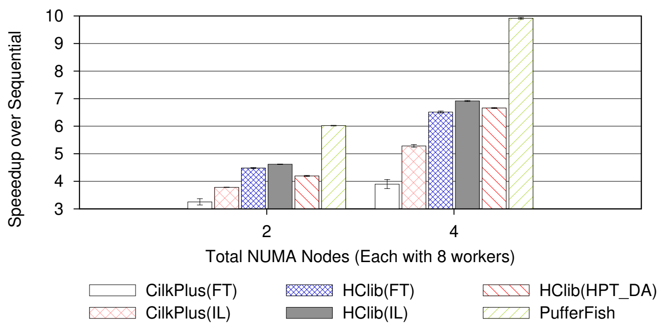
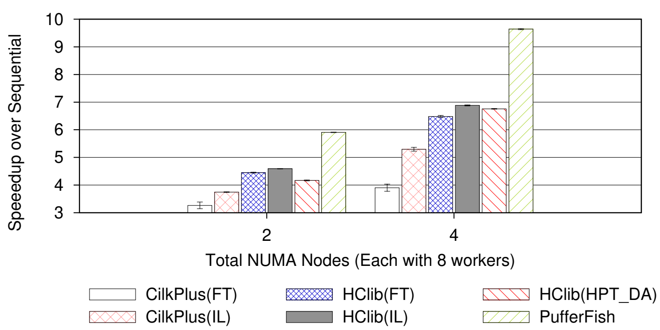


Fig. 9. Speedup of CilkSort



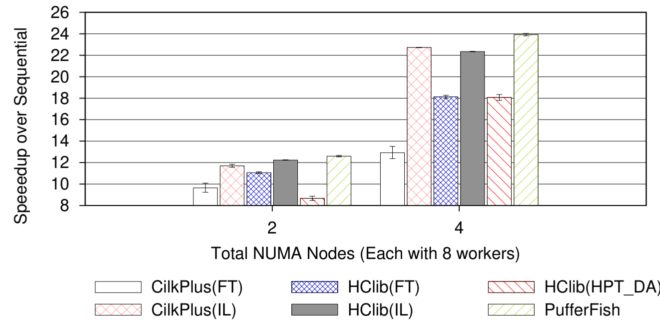
(a) LULESH



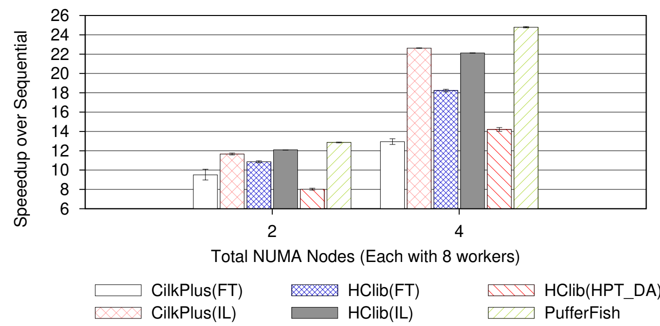
(b) LULESH-ir

Fig. 10. Speedup of LULESH with regular and irregular execution DAGs

loaded from its official github repository with the commit id `ab310a0`. We used -O3 flag for compiling our benchmarks. The benchmarks were run on a 32-core AMD EPYC 7551 processor. Maximum and minimum frequency of this processor is 2GHz and 1.2GHz, respectively. We preserved the default settings of the system with the CPU governor policy set to `ondemand`. This machine had a total of 64GB of RAM. The operating system was Ubuntu 18.04.3 LTS. Each implementations were executed ten times, and we report the mean of the execution time, along with a 95% confidence interval.

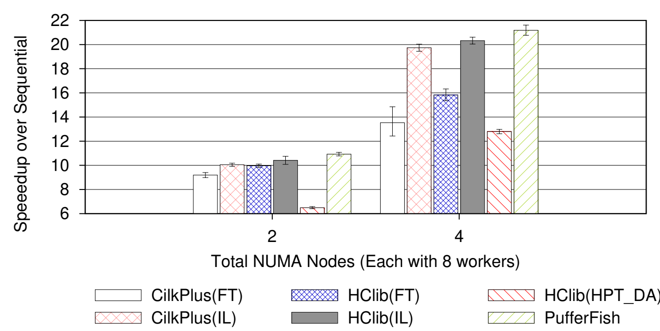


(a) SOR

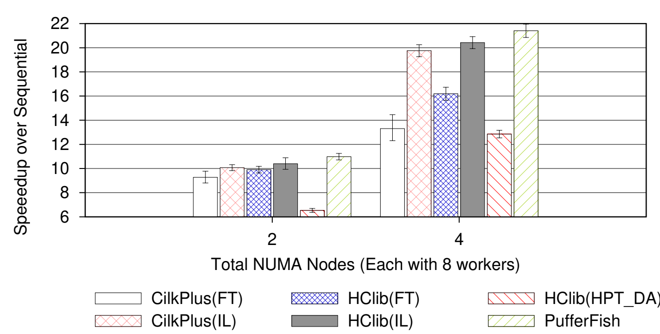


(b) SOR-ir

Fig. 11. Speedup of SOR with regular and irregular execution DAGs



(a) SRAD



(b) SRAD-ir

Fig. 12. Speedup of SRAD with regular and irregular execution DAGs

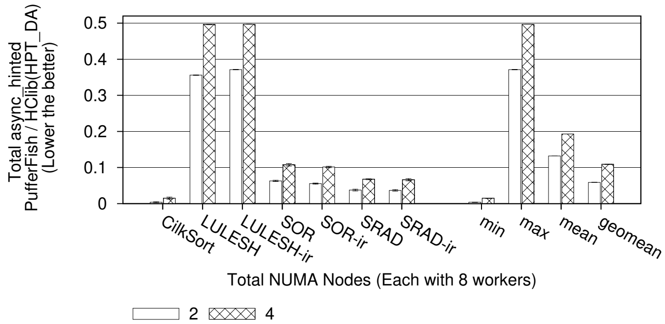


Fig. 13. Total number of **async_hinted** created in `PufferFish` relative to `HClib (HPT_DA)`

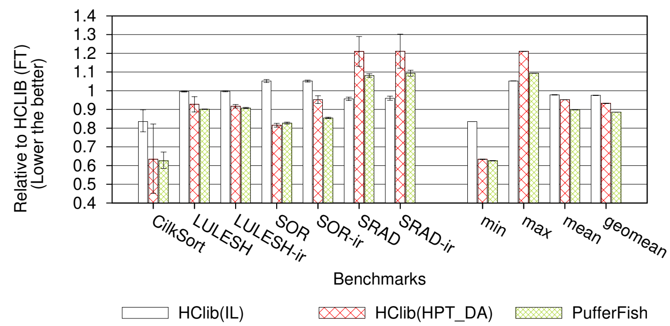


Fig. 14. L2-cache misses relative to `HClib` by using four NUMA nodes

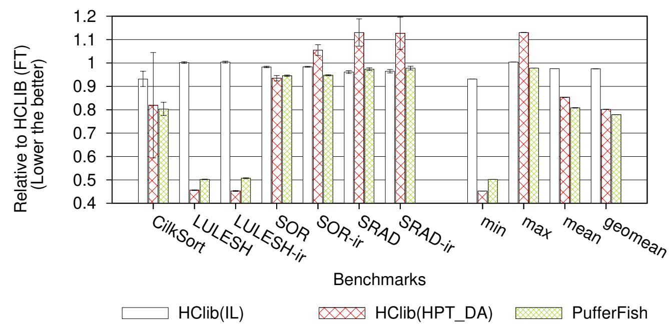


Fig. 15. L3-cache misses relative to `HClib` by using four NUMA nodes

VI. Experimental Evaluation

A. Performance Analysys

Figure 9, Figure 10, Figure 11, and Figure 12 shows the speedup obtained by executing different implementations of CilkSort, LULESH, SOR, and SRAD, respectively, over two and four NUMA nodes. Geomean speedup obtained in `CilkPlus (FT)`, `CilkPlus (IL)`, `HClib (FT)`, `HClib (IL)`, `HClib (HPT_DA)`, and `PufferFish` over the `Sequential` implementation by using four NUMA nodes was 8.8×, 13.3×, 11.2×, 14.2×, 11×, and 17×, respectively. A similar trend holds even with two NUMA nodes. `PufferFish` wins over each implementation in both the NUMA settings. As expected, interleave allocation policies in `CilkPlus` and `HClib` performs significantly better the first-touch policies in respective implementations. As both these implementations use random work-stealing, spreading the physical pages of the memory in round-robin over the NUMA nodes improves the performance. `HClib` wins over `CilkPlus` implementation in both these allocation policies.

An important point to observe is that the `HClib (HPT_DA)` performs even poor than `HClib (FT)` and `HClib (IL)`. Recall from Section V, `HClib (HPT_DA)` uses the same programming interface as in `PufferFish` but uses the default hierarchical work-stealing in `HClib`. As described in Section II-B, it is challenging to use `async_at` interface in `HClib` in recursive benchmarks. Hence, for `HClib (HPT_DA)` we used **async_hinted** with data-affinity hints to map the tasks to optimal `place` in HPT, but used `HClib`'s default hierarchical work-stealing. We carried out this experiment to study the performance of `PufferFish`'s novel hierarchical work-stealing implementation (Section IV-C) that internally uses hierarchical elastic tasks (Section IV-D). This analysis clearly shows that the `PufferFish` parallel programming model is a simple and scalable solution for NUMA-aware work-stealing. `HClib (HPT_DA)` show poor performance due to worker starvation (Section IV-C).

B. Analysys of Hierarchical Elastic Tasks (HET)

`PufferFish` uses HET implementation (Section IV-D) of **async_hinted** for reducing the context switches for task execution at the leaf `place` (L3 in our case). Figure 13 shows the ratio by which `PufferFish` was able to reduce the number of **async_hinted** in each benchmark as compared to `HClib (HPT_DA)`. Both these implementation uses the same programming interface, but different work-stealing implementations. We can observe that HET in `PufferFish` reduces the task creation by up to 95% and 90% in two and four NUMA node settings. Its effect in LULESH and LULESH-ir is minimum because this benchmark has pipeline parallelism, and it generates coarse granular tasks at each stage in the pipeline. HET shows better performance with the benchmarks having fine granular tasks.

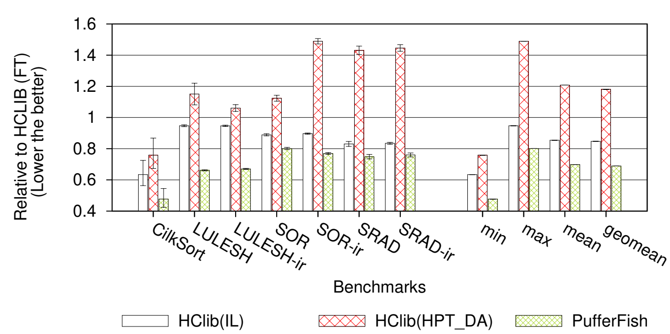
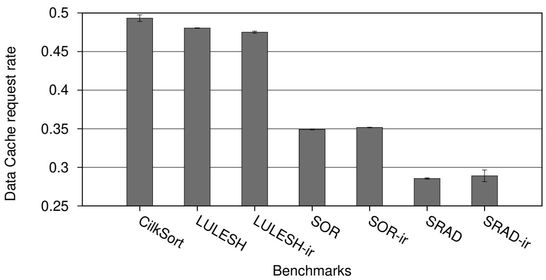
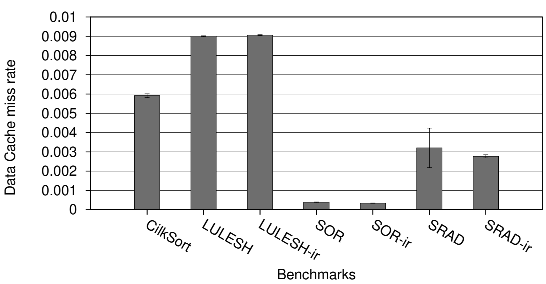


Fig. 16. Package energy relative to `HClib` by using four NUMA nodes



(a) Data-cache request rate tells how data intensive is an application



(b) Data-cache miss rate gives a measure of how often it was necessary to get cache lines from higher levels of memory hierarchy

Fig. 17. Memory access pattern of different benchmarks by using `PufferFish` over four NUMA nodes

C. Hardware Performance Counter Results

To further understand the benefits of `PufferFish` on overall system performance, we used the `LIKWID` APIs [29] to study the cache misses and energy savings in `PufferFish`. Figure 14, Figure 15, and Figure 16, shows the L2-cache misses, L3-cache misses, and energy savings, respectively, in `HClib (IL)`, `HClib (HPT_DA)` and `PufferFish` relative to `HClib (FT)` by using four NUMA nodes. Overall, `PufferFish` achieves its goal better than `HClib (IL)` and `HClib (HPT_DA)` across all these three results.

D. Memory Access Pattern in Benchmarks

During our speedup analysis (Section VI-A), we found `PufferFish` was giving slightly lesser speedup in CilkSort and variants of LULESH as compared to the SOR and SRAD variants. For understanding this result, we used `LIKWID` to profile the memory accesses pattern of all the benchmarks using `PufferFish` execution over four NUMA nodes. We calculated the Data Cache Request Rate (DCRR) and Data Cache Miss Rate (DCMR), the result for the same is shown in Figure 17(a) and Figure 17(b) respectively. DCRR tells how data-intensive an application is, and DCMR measures how often it was necessary to get cache lines from higher levels of the memory hierarchy. CilkSort's DCRR and DCMR are on the higher side compared to SOR and SRAD as it moves a huge chunk of arrays across the NUMA memory domains. These values are smaller for SOR and SRAD variants as they operate on a single array in iterations. Hence, they are already hugely benefiting from the temporal locality. This is the reason they achieve higher speedup even with `CilkPlus (IL)`, and `HClib (IL)`. LULESH variants exhibit higher DCRR and DCMR. This is due to the design of this benchmark. LULESH also operates in iterations, but each iteration has 24 parallel regions operating on a wide range of arrays (total 78 arrays allocated using **`numa_alloc_blockcyclic`** API). Hence, the locality benefit in LULESH is lesser than other benchmarks, thereby resulting in lesser speedups.

VII. Related Work

The performance of a memory-bound task-based parallel program can be improved on a NUMA system by following a two-step procedure: *Step-1*) spreading the physical pages of the entire memory that a program would access over all the NUMA nodes, and *Step-2*) scheduling the tasks on the NUMA node that contains the physical pages of the memory accessed by this task. Libraries such as `libnuma` [24] on Linux provide a rich set of APIs to achieve the Step-1. However, Step-2 is challenging to achieve due to the dynamic creation of tasks. Asking the programmer to perform the top-level partitioning of tasks across NUMA nodes is a popular choice for achieving the Step-2 [22], [23], [1], [10], [13], [12]. This step can also be achieved automatically in an iterative application by profiling its iterations and using these profile results to map tasks in remaining iterations on appropriate NUMA node [9], [28]. Once the top-level task partitioning is done in Step-2, a common approach to preserving the task's locality is by using some flavor of hierarchical work-stealing. Thieves in HotSLAW [30] steal a small number of tasks or half of the victim's deque tasks based on how near or how far away the victim is from this thief in NUMA hierarchy. Thieves in [31] steal directly from victims under the same NUMA hierarchy and rely on a team leader for pursuing remote steals to reduce the cost of remote latency. NUMA work-stealing in Cilk [12], and Intel TBB [13] used a combination of worker local deque and *mailbox* [32]. Steals within a NUMA domain are performed from worker's local deque, whereas the mailbox is used to push tasks with affinity to a different NUMA node. User-specified top-level partitioning of tasks usually works well with programs exhibiting regular execution DAG, but it is challenging to evenly partition the dynamically unfolding tasks in a program exhibiting irregular execution DAG.

Huang et al. proposed a data-affinity clause with OpenMP tasks [33] as a natural way to map the tasks than HPT to overcome this limitation. This approach was further studied in [34], [14], and is now incorporated in the latest OpenMP 5.0 standards [15]. A common limitation across all these studies is they don't propose a scalable NUMA-aware work-stealing implementation for these tasks. `PufferFish` overcomes this limitation by introducing a new **`async–finish`** programming model that integrates the data-affinity hints with an HPT implementation in `HClib` and supports a scalable hierarchical work-stealing implementation.

Sbîrlea et al. introduced elastic tasks for improving the locality of random work-stealing [16]. They introduced a new **`async`** API that required input from the user about the amount of work and parallelism in that task. This information was then used by the runtime to run on a single worker or expand to take over multiple workers based on the system workload. `PufferFish` took inspiration from them and implements Hierarchical Elastic Tasks (HET) that do not require any extra input from the user. HET activates itself hierarchically based on the level of a `place` it is attached to in an HPT.

Recently proposed ADWS [35] expects the programmer to specify the amount of work in each task and uses this information for deterministic task allocation over workers. It further uses a hierarchical work-stealing for improving the NUMA locality. Drebes et al. utilized the data dependency information in a data-flow programming model to decide the NUMA aware task placement [36], [37]. `PufferFish` is somewhat similar to their work in the sense that it relies on explicit data-affinity hints instead of implicitly derived hints.

VIII. Conclusion

Multicore processors based on NUMA architecture are now mainstream and poses enormous challenges in achieving a good performance in memory-bound task-parallel programs. Existing solutions rely on programmer-based approaches for distributing the tasks evenly across all NUMA nodes. This architecture-specific optimal partitioning is hard to achieve in a dynamically unfolding task-based parallel programming model. An orthogonal approach to solve this problem is assigning data-affinity hints with the parallel tasks instead of programmer specified task partitioning. In this paper, we designed and implemented a new **`async–finish`** programming model for specifying data-affinity hints. It builds over existing solutions, but significantly improve the performance by using a novel NUMA-aware work-stealing runtime. Our empirical results demonstrate that we can achieve better performance on a NUMA system than traditional approaches for task parallelism.

References

[1] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *LCPC '10*, 2010, pp. 172–187.

[2] D. Lea, "A Java Fork/Join framework," in *JAVA*, 2000, pp. 36–43.

[3] "OpenMP API, version 4.5," accessed February 2019. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[4] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar, "HabaneroUPC++: A compiler-free PGAS library," in *PGAS*, 2014, pp. 5:1–5:10.

[5] J. Reinders, *Intel Threading Building Blocks*, 1st ed. O'Reilly & Associates, Inc., 2007.

[6] A. D. Robison, "Composable parallel patterns with Intel Cilk Plus," *Computing in Science and Engg.*, vol. 15, no. 2, p. 66–71, 2013.

[7] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, Sep. 1999.

[8] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Optimizing data locality for fork/join programs using constrained work stealing," in *SC '14*, 2014, pp. 857–868.

[9] Q. Chen, M. Guo, and Z. Huang, "CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures," in *ICS'12*, 2012, p. 163–172.

[10] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar, "SLAW: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems," in *PPoPP '10*, 2010, pp. 341–342.

[11] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *IPDPS '13*, 2013.

[12] J. Deters, J. Wu, Y. Xu, and I. A. Lee, "A NUMA-aware provably-efficient task-parallel platform based on the work-first principle," in *IISWC'18*, 2018, pp. 59–70.

[13] Z. Majo and T. R. Gross, "A library for portable and composable data locality optimizations for NUMA systems," in *PPoPP'15*, 2015, p. 227–238.

[14] C. Terboven, J. Hahnfeld, X. Teruel, S. Mateo, A. Duran, M. Klemm, S. L. Olivier, and B. R. de Supinski, "Approaches for task affinity in OpenMP," in *IWOMP'16*, N. Maruyama, B. R. de Supinski, and M. Wahib, Eds., 2016, pp. 102–115.

[15] OpenMP ARB, *OpenMP Application Programming Interface Version 5.0*, November 2018. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[16] A. Sbîrlea, K. Agrawal, and V. Sarkar, "Elastic tasks: Unifying task parallelism and SPMD parallelism with an adaptive runtime," in *Euro-Par'15*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Springer Berlin Heidelberg, 2015, pp. 491–503.

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

[18] S. Imam and V. Sarkar, "Habanero-Java Library: A Java 8 framework for multicore programming," in *PPPJ*, 2014, pp. 75–86.

[19] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu, "Work-stealing without the baggage," in *OOPSLA*, 2012, pp. 297–314.

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

[21] Yi Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *IPDPS*, 2009, pp. 1–12.

[22] K. Ebcioglu, V. Saraswat, and V. Sarkar, "X10: an experimental language for high productivity programming of scalable systems." Citeseer, 2005, pp. 45–52.

[23] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[24] A. Kleen, "A NUMA api for linux," *Novel Inc*, 2005.

[25] V. Kumar, "PufferFish: NUMA-aware work-stealing library using elastic tasks (Artifact)," 2020. [Online]. Available: https://github.com/hipec/pufferFish/archive/v1.0.zip

[26] I. Karlin, "Lulesh programming model and performance ports overview," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2012.

[27] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009, pp. 44–54.

[28] Q. Chen, M. Guo, and H. Guan, "LAWS: Locality-aware work-stealing for multi-socket multi-core architectures," in *ICS'14*, 2014, p. 3–12.

[29] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *ICPPW'10*. IEEE, 2010, pp. 207–216.

[30] S. jai Min, C. Iancu, and K. Yelick, "Hierarchical work stealing on manycore clusters," in *PGAS'11*, 2011.

[31] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins, "Scheduling task parallelism on multi-socket multicore systems," in *ROSS '11*, 2011, p. 49–56.

[32] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *SPAA '00*, 2000, p. 1–12.

[33] L. Huang, H. Jin, L. Yi, and B. Chapman, "Enabling locality-aware computations in OpenMP," *Sci. Program.*, vol. 18, no. 3–4, p. 169–181, 2010.

[34] A. Muddukrishna, P. A. Jonsson, and M. Brorsson, "Locality-aware task scheduling and data distribution for OpenMP programs on NUMA systems and manycore processors," *Scientific Programming*, 2015.

[35] S. Shiina and K. Taura, "Almost deterministic work stealing," in *SC '19*, 2019.

[36] 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'16*, 2016, p. 125–137.

[37] 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 TACO*, vol. 11, no. 3, 2014.