# COWS for High Performance

Cost Aware Work Stealing for Irregular Parallel Loops

PRASOON MISHRA, Indian Institute of Technology Madras, India
V. KRISHNA NANDIVADA, Indian Institute of Technology Madras, India

Parallel libraries such as OpenMP distribute the iterations of parallel-for-loops among the threads, using a programmer-specified scheduling policy. While the existing scheduling policies perform reasonably well in the context of balanced workloads, in computations that involve highly imbalanced workloads it is extremely non-trivial to obtain an efficient distribution of work (even using non-static scheduling methods like dynamic and guided). In this paper, we present a scheme called COst aware Work Stealing (COWS) to efficiently extend the idea of work-stealing to OpenMP.

In contrast to the traditional work-stealing schedulers, COWS takes into consideration that (i) not all iterations of a parallel-for-loops may take the same amount of time. (ii) identifying a suitable victim for stealing is important for load-balancing, and (iii) queues lead to significant overheads in traditional work-stealing and should be avoided. We present two variations of COWS: WSRI (a naive work-stealing scheme based on the number of remaining iterations) and WSRW (work-stealing scheme based on the amount of remaining workload). Since in irregular loops like those found in graph analytics, it is not possible to statically compute the cost of the iterations of the parallel-for-loops, we use a combined compile-time + runtime approach, where the remaining workload of a loop is computed efficiently at runtime by utilizing the code generated by our compile-time component. We have performed an evaluation over seven different benchmark programs, using five different input datasets, on two different hardware across a varying number of threads; leading to a total of 275 number of configurations. We show that in 225 out of 275 configurations, compared to the best OpenMP scheduling scheme for that configuration, our approach achieves clear performance gains.

CCS Concepts: • **Software and its engineering** → *Compilers*; *Runtime environments*.

**ACM Reference Format:**
Prasoon Mishra and V. Krishna Nandivada. 2023. COWS for High Performance: Cost Aware Work Stealing for Irregular Parallel Loops. 1, 1 (November 2023), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

With multicore systems going mainstream, efficient parallel computing has become a necessity rather than a choice. One common pattern seen in modern parallel applications is loop-level parallelism. Almost all the modern task-parallel languages, like OpenMP [27], Cilk [1], Chapel [6], HJ [5], X10 [7], and others, have built-in support for parallel-for-loops, which are used by the programmers to parallelize the main computing tasks of the applications. Thus, the performance of the parallel applications typically depends on that of the parallel-for-loops, which in turn depends on how the iterations of the loop are distributed among the runtime threads; this problem is commonly known as the loop-scheduling problem. One of the common optimization techniques of loop-scheduling is to ensure that load-imbalance between the threads is minimized.

Authors' addresses: Prasoon Mishra, Indian Institute of Technology Madras, Chennai, India, prasoon@cse.iitm.ac.in; V. Krishna Nandivada, Indian Institute of Technology Madras, Chennai, India, nvk@iitm.ac.in.


XXXX-XXXX/2023/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
1 #pragma omp parallel
2 {
3  #pragma omp for schedule(SCHEDULING_METHOD)
4  for (int i = 0; i < g->N; i++) {
5   node* cur = elem_at(&g->vertices, i);
6   p = cur->data;
7   for (int j = 0; j < cur->degree; j++) {
8    node* neighbor=elem_at(&cur->neighbors,j);
9    neighbor_p = neighbor->data;
10   /*Some work involving p and neighbor_p*/}}}
```

Fig. 1. A typical pattern found in many IMSuite kernels.

In regular loops, where work done in each iteration is comparable and mostly utilizes a regular data access pattern, achieving a good load balance is relatively easy. However, when with irregular loops, the loop scheduling problem becomes very challenging as illustrated in the following example.

Figure 1 shows a typical pattern (in C+OpenMP) found in many IMSuite [13] kernels and many graph analytics kernels. At Line 1, the `omp parallel` pragma creates a team of threads, to concurrently execute the parallel-region (Lines 2-10). At Line 3, the `omp for` pragma defines a work-sharing construct (parallel-for-loop) that distributes the iterations of the parallel-for-loop among the team of threads (using a user chosen scheduling method). Thus, in Line 4, the parallel-for-loop has `g->N` number of independent iterations to be distributed among the team of threads. Even though every iteration of the loop executes the same code, the time to execute each iteration depends on the serial loop at Line 7. Since the value of `cur->degree` is input dependent and varies with different values of `i`, it becomes hard to identify an optimal schedule for such parallel-for-loops, especially when the value of the `degree` field of the vertices varies significantly. For example, for the YouTube dataset (discussed in Section 6) the degree of vertices varies between 1 to 28800. This is a typical case with all the power-law graphs [37], a highly important type of input; some examples of power-law graphs include: twitter (and other social media) connection graphs, citation networks, web-graphs, and so on [23].

Loop-scheduling is a NP-hard problem and there have been various approaches proposed in the past [4, 24, 25] to tackle this problem. Over the past decade or so, there have been many performance critical applications using irregular parallel loops, which bring in additional challenges for loop-scheduling. Some of the current approaches to derive performance in application with irregular parallel loop include static scheduling [27], loop-chunking [25], deep chunking [29], dynamic and guided scheduling [27], work-stealing [1], and so on. In static scheduling, loop-chunking and deep-chunking, the iterations of the loop are distributed among the threads, before the loop starts executing. While such a scheme incurs very low overheads, it suffers from the fact that once the iterations are assigned to any thread, they are not reassigned to another thread, even if the latter thread has finished executing its assigned list of iterations. This may cause a fair amount of load-imbalance. Recently, Booth and Lane [3] presented a case for a randomly chosen victim based scheme of work-stealing scheduling (with dynamic chunk-size) for OpenMP and show that its performance is comparable to that of dynamic/guided schemes of OpenMP. These dynamic and guided scheduling schemes of OpenMP and the popular work-stealing scheme allow the iterations to be distributed at runtime, they suffer from multiple drawbacks: (1) they incur a fair amount of overhead to maintain the work-queues (for example, Nandivada et al. [25] show that the work-stealing runtime performs much worse than the simple block-chunked code), (2)

identifying the optimal block-size is non-trivial for irregular loops, (3) once a set of iterations have been assigned to a thread, they cannot be reallocated based on the actual workload.

To address these issues, in this paper, we propose a scheme that efficiently extends the idea of work-stealing to OpenMP called COst aware Work Stealing (COWS). COWS gives us the best of both worlds: the low overheads of static scheduling and dynamic load balancing of workstealing.

We present two variations of COWS: work-stealing based on remaining iterations (WSRI) and work-stealing based on remaining workload (WSRW). For WSRI, we maintain the remaining iterations for each thread in an efficient data-structure. For WSRW, we use a mixed compile-time and runtime approach: the compile-time analysis emits application-specific code to efficiently estimate the workload (at runtime), and the remaining workload is computed at runtime by executing this code. In WSRI and WSRW, our modified work-stealing scheduler uses the remaining iterations and remaining workload, respectively, to choose a suitable victim. Even though we present the idea of COWS in the context of OpenMP, it can be used in other parallel languages (such as Cilk, X10, Chapel, and so on) that support parallel-for-loops.

**Our Contributions:**

• We propose a new technique called COWS for load balancing irregular parallel loops. COWS supports two different schemes that differ on how the victims are chosen: work-stealing based on remaining iterations (WSRI) and work-stealing based on remaining workload (WSRW).

• We propose a queue-free cost-aware work stealing algorithm to choose the victim for work-stealing. This is based on an efficient scheme to maintain the remaining iterations for each thread.

• We have implemented the static compiler component of COWS (relevant for WSRW) in the IMOP compiler framework and the runtime component (relevant for both WSRI and WSRW) in libgomp a shared library for OpenMP in GCC. We have studied the performance of COWS on seven popular shared memory kernels running on two different hardware systems and show that COWS performs significantly better than the cyclic, static, dynamic, guided, and WSR (work-stealing with random victim selection) schedules, for power-law graphs. We have also found that in the context of roughly-regular graphs and COWS gives reasonable performance improvement over the rest.

The rest of the paper is organized as follows. We give a brief background about some of the important relevant concepts in Section 2. In Section 3, we describe our proposed cost aware work stealing (COWS) scheme. In Section 4, we discuss four new optimizations to improve the performance of the proposed COWS scheme. In Section 5, we discuss some salient features of our proposed scheme. We present an elaborate evaluation of our scheme in Section 6. We discuss the related work in Section 7 and conclude in Section 8.

## 2 Background

**Scheduling schemes in OpenMP.** OpenMP supports four scheduling policies (static, dynamic, guided, and runtime) to map an iteration `i` of parallel-for-loop (with `N` iterations) to one of the `T` threads. (i) `static` schedule: scheduling method specified as `static[, ch]`. The optional argument `ch` may be passed by the programmer to specify the chunk size and the iteration `i` gets mapped to the thread `(i / ch)%T`. We term a special case of static scheduling when `ch` = 1, as `cyclic` chunking. When the option `ch` is not passed, it is set to a default value `N/T`; we use the term `static` schedule to denote this default scheduling policy. (ii) `dynamic` schedule: scheduling method is specified as `dynamic[, ch]`. Every time a thread is idle, it is assigned at most `ch` (default value = 1) iterations. (iii) `guided` schedule: scheduling method is specified as `guided[, ch]`. Every time a thread is idle, it is assigned max(`ch`, remaining-iterations/number-of-idle-threads) number of iterations. (iv) `runtime` schedule: scheduling policy is inferred at runtime by reading the environment variable (`OMP_SCHEDULE`), which in turn must be set to one of the above three policies. To implement our

```
#pragma omp for schedule (runtime)
  for (all-iterations) { loop-body }
                    (a)

// assign a list of iterations to each thread.
GOMP_parallel_loop_runtime();
do{
 for (all-assigned-iterations) { loop-body }
 GOMP_loop_runtime_next(); //next iters list
} while (iterations-are-available);
                    (b)
```

Fig. 2. An OpenMP parallel-for-loop (a) and sketch of its translation by GCC (b).

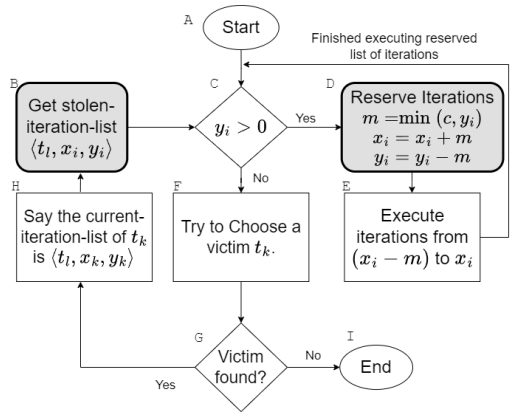



Fig. 3. Overall control flow of thread $t_i$, whose current-iteration-list is $\langle t_j, x_i, y_i \rangle$ and reservation size is $c$.

proposed work-stealing approach, the programmer has to state the schedule clause as `runtime` and set `OMP_SCHEDULE` to one of the two proposed schemes: `WSRW` or `WSRI`.

**Translation of parallel-for-loops.** Figure 2 shows how the existing GCC compiler translates a parallel-for-loop with runtime-schedule. The translated code is executed by each thread. We piggy-back on this translation scheme to invoke our proposed work-stealing algorithms (in Section 3.4.2).

**Work Stealing.** In work-stealing scheduling, each thread is initially given a set of iterations to execute. When a thread runs out of work of its own, it identifies a victim and steals some work from it. Typically the work (for example, the subset of iterations of the parallel-for-loop) assigned to (or stolen by) each thread is stored in a queue.

## 3 Cost Aware Work Stealing

The traditional work-stealing schedulers, despite their advantages, suffer from multiple issues, especially when applied to parallel-for-loops: (i) the schedulers are unaware of the workload of different tasks and consider all the tasks are equal, (ii) choosing the suitable victim for stealing to improve load-balancing is challenging, (iii) the overhead of maintaining the queue can be expensive, and (iv) these schedulers have been designed to efficient work on a list of explicitly created individual tasks (and not iterations of a loop like OpenMP parallel-for-loops). In this section, we present the details of our proposed COst aware Work Stealing (COWS) scheme that addresses these issues. Our scheduling scheme targets the parallel-loops in irregular task parallel programs, whose iterations are distributed among the different runtime threads.

The COWS scheme does not maintain any global queue. Instead, the iterations of the for-loops are initially distributed among all the runtime threads (say, total $T$ number of threads). For efficiency, we assume this initial distribution is done using a cyclic distribution (iteration $i$ is assigned to a thread with thread-id $i\%T$). For ease of explanation, we use the subscript notation $t_j$ to denote the thread with thread-id $j$. Like typical work-stealing, once a thread runs out of work (iterations assigned to it), the thread tries to steal work from other threads. In this process, our scheme tries to steal from that thread whose remaining workload is maximum. We now detail our scheme below.

### 3.1 Efficiently Maintaining the Remaining Workload

Considering the overheads of maintaining queues to store the tasks assigned to any thread, we use two key insights: (i) the initial list of iterations assigned to any thread $t_i$ need not be stored and can be inferred any time by using the thread id $i$, the total number of threads $T$, and the total number of iterations $N$; for example, say $N = 10$, $T = 3$, then the initial list of iterations assigned to $t_0$ are $[0, 3, 6, 9]$. For any thread, we call this initial list of iterations to be executed as

initial-iteration-list. (ii) At any point of time during execution, a thread may be executing iterations from its initial-iteration-list or iterations stolen from the initial-iteration-list of another thread. The current-iteration-list for the $i^{th}$ thread $t_i$ can be represented as a three tuple $\langle t_j, x_i, y_i \rangle$, where $t_j$ represents the $j^{th}$ thread, $x_i$ indicates the index of the iteration in the initial-iteration-list of $t_j$ that $t_i$ has to execute, and $y_i$ indicates the number of remaining iterations to be executed by $t_i$ from the initial-iteration-list of $t_j$ starting from $x_i$.

### 3.2 Control flow of a thread

Figure 3 depicts the flowchart of the overall control flow of any thread. For efficiency, we allow a parameter $c$ (detailed in Section 5) denoting the reservation size; each thread reserves $c$ iterations at a time from its current-iteration-list and executes them, with a guarantee that no thread can steal from these reserved iterations. A thread $t_i$, with current-iteration-list $\langle t_j, x_i, y_i \rangle$, starts by checking if it has a non-zero number of remaining iterations ($y_i$).

If $y_i > 0$, the thread first atomically reserves at most $c$ iterations from the current-iteration-list (box D). Then it executes those $c$ iterations sequentially. This scheme ensures that no other thread can steal the reserved iterations while $t_i$ is executing these iterations. In the flowchart, greyed boxes with the bold boundaries indicate the critical sections.

If $y_i \leq 0$, $t_i$ will become a thief and try to choose a victim thread $t_k$. We will discuss about the different schemes to choose the suitable victims later in this section. If $t_i$ can find a victim $t_k$, then $t_i$ will try to atomically steal half the remaining workload from the current-iteration-list of $t_k$ (box B). We steal half the iterations as it leads to the smallest critical path in terms of the number of remaining iterations. This stealing includes (atomically) setting appropriate values in the current-iteration-list of both the victim and the thief. Note that while the thief $t_i$ is choosing a victim and trying to set its current-iteration-list, the victim continues to execute its iterations and may have proceeded to complete more iterations than what $t_i$ had observed in box F. And hence, before proceeding to execute the stolen iterations, the thief must check if the stolen-iteration-list has non-zero iterations ($y_i > 0$). More details about this modified work-stealing algorithm can be found in Section 3.5. Each thread continues this cycle, as long as victim threads are found.

**Example.** Assume that there are 10 iterations in a parallel-for-loop. Figure 4 shows how different threads maintain their iteration lists, if there are 3 threads at runtime, and $c = 1$. The initial-iteration-list is the same as the current-iteration-list in the starting (row 3). As each thread has some iterations to execute, they will reserve $c$ iterations from the current-iteration-list. Row 4 shows the updated current-iteration-lists after the first reservation. Suppose $t_2$ completes all the remaining iterations while other threads are still executing their first iteration; Row 5 shows the current-iteration-lists. Now $t_2$ becomes a thief and will try to choose a victim. Suppose that it chooses $t_0$ as the victim and successfully steals work (half of the remaining iterations = 1 iteration) from it. Row 6 shows the current-iteration-lists after the stealing. Supposing that no more stealing happens, and the threads complete their assigned iterations, Row 7 shows the final values for the current-iteration-lists.

In work-stealing, choosing a suitable victim is an important consideration for obtaining highly performant execution. Another equally important consideration is the time taken to choose the victim, as the overheads for the same, can overshadow the gains due to work-stealing. For a scheme that chooses the victim randomly, its efficiency takes a big hit due to the possibly large overheads arising out of a large number of times the thieves may have to steal the iterations. We call such a scheme WS-Random (WSR, in short).

We now discuss two strategies to choose the victims such that (i) we minimize the number of steals, and (ii) the overheads for victim selection are minimal. Both of our strategies are inspired by the simple intuition that a thief should choose a victim with the maximum amount of remaining work, which in turn is likely to reduce the number of steals and improve the overall performance.

| 1. | | $t_0$ | $t_1$ | $t_2$ |
|---|---|---|---|---|
| *After initial cyclic distribution.* | | | | |
| 2. | List of iterations | [0, 3, 6, 9] | [1, 4, 7] | [2, 5, 8] |
| 3. | initial-iteration-list | $\langle 0, 0, 4\rangle$ | $\langle 1, 0, 3\rangle$ | $\langle 2, 0, 3\rangle$ |
| *After each thread has reserved c iterations, for itself.* | | | | |
| 4. | current-iteration-list | $\langle 0, 1, 3\rangle$ | $\langle 1, 1, 2\rangle$ | $\langle 2, 1, 2\rangle$ |
| $t_2$ *finished all the initially assigned iterations, other threads still executing iteration#1.* | | | | |
| 5. | current-iteration-list | $\langle 0, 1, 3\rangle$ | $\langle 1, 1, 2\rangle$ | $\langle 2, 3, 0\rangle$ |
| *After* $t_2$ *has stolen one iteration from* $t_1$. | | | | |
| 6. | current-iteration-list | $\langle 0, 1, 2\rangle$ | $\langle 1, 1, 2\rangle$ | $\langle 0, 3, 1\rangle$ |
| *All threads finished assigned iterations; no more stealing.* | | | | |
| 7. | current-iteration-list | $\langle 0, 3, 0\rangle$ | $\langle 1, 3, 0\rangle$ | $\langle 0, 4, 0\rangle$ |

Fig. 4. Illustration of how the threads maintain their iteration lists.

### 3.3 Remaining Iterations as the Estimate for Remaining Work

Since we are only focused on distributing the iterations of a parallel-for-loop among the threads, one of the most intuitive estimates for measuring the workload remaining with any thread is the number of remaining iterations assigned to that thread. To be able to use this metric, a thief needs to be able to obtain the remaining number of iterations of every possible victim. This is very easy to obtain: the thief can iterate over the current-iteration-list of each thread (typically the number of threads is a small number) to obtain the victim with the maximum number remaining-iterations. We call such a scheme WS-Remaining Iterations (WSRI, in short).

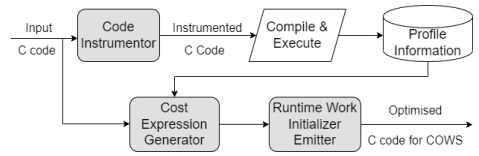
### 3.4 Using the Remaining Work As the Cost Estimate

When a team of threads are executing a parallel-for-loop, the metric of the remaining-number-of-iterations of thread may not be an accurate estimate of the remaining workload currently assigned to the thread. This is especially true in the context of irregular task parallel programs, where the cost of each iteration can vary significantly. To address such a challenge we propose a work-stealing scheme based on the remaining workload (WSRW, in short).

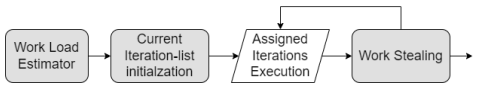
Statically estimating the cost of the iterations during compilation time may not be possible as the iteration code may be input-dependent. For example, for the code snippet shown in Figure 1, each iteration `i` of the parallel-loop iterates over the neighbors of the $\texttt{i}^{th}$ node. Thus, the cost of each iteration `i` depends on the degree of the $\texttt{i}^{th}$ node and cannot be estimated statically. To address such issues, inspired by prior works that recognize such a difficulty [29, 33], we propose to use a mixed static + dynamic technique, where we first estimate the cost of each iteration as a function parameterised over some input variable. In the previous example, the cost of iteration `i`, can be expressed as a function of `elem_at(&g->vertices, i)->degree`. We take such parametric cost-expressions and use them to compute the actual cost at runtime. Figure 5 shows the block diagram of our overall process, consisting of a compile-time component and a runtime component.

*3.4.1 Compile-time Component.* The input OpenMP C program is first profiled (instrumented and executed) to obtain the costs of every input independent part of the code; we use a technique similar to that of Shrivastava and Nandivada [33]. The next phase of compilation (shown in Figure 5a) has two main steps: (i) Cost expression generator and (ii) Runtime work initializer emitter.

*Cost expression generator.* We use the profile generated costs to generate cost-expressions for both input dependent and independent parts of the code. This cost-expression generator uses the scheme of Shrivastava and Nandivada [33, Section 3.1, Figure 7] to generate the cost expression (`CostExpr`) for the body of each parallel-for-loop; we give a brief summary of the same in Appendix A. For example, in the code shown in Figure 1, Lines 5-6, and 8-10 are input independent. Say the cost of input independent statement at line $i$ is given by $C_i$. Similarly, for the for-loop, say $C_{7-1}$ gives




(a) Compile-time component.




(b) Runtime-time component.

Fig. 5. Overall technique: the grey boxes are the main components of our technique.

```
1  int tid = omp_get_thread_num();
2  long chunk = loopBound/T + 1;
3  int counter = 0;
4  long result = 0;
5  TArr[tid]=realloc(TArr[tid],
6                    chunk*sizeof(long));
7  #pragma omp for schedule(static,1)
8  for (int i=0; i<loopBound; i++){
9    result += CostExpr;
10   TArr[tid][counter++] = result;}
```

Thread local

Fig. 6. Runtime workload-estimator code, emitted just before a parallel-for-loop, whose for-loop is of the form `for(i=0;i<loopBound;i++){...}`.

the cost of the initialization statement, $C_{7-2}$ gives the cost of the predicate, and $C_{7-3}$ gives the cost of the increment statement. Then the cost expression returned by the `getWL` function of Shrivastava and Nandivada [33] for the body of the parallel-for-loop would be $C_5$+$C_6$+$C_{7-1}$+ `(node *)elem_at(&g->vertices, i)->degree*` ($C_{7-2}$ + $C_8$ + $C_9$ + $C_{10}$ + $C_{7-3}$). In case, the `getWL` function fails to obtain an accurate cost-estimate for the iterations of a parallel-for-loop (for example, in case of iterations contain a while-loop), then for that parallel loop, we abort the WSRW scheme, and fall back to WSRI.

*Runtime workload-estimator emitter.* As we had discussed before, our goal is to emit some small code that can estimate the remaining-workload of each iteration at runtime. Since a program may contain multiple parallel-for-loops and the workloads (or cost, in short) of their iterations across different loops may vary, we may have to emit the workload estimating code for each parallel-for-loop separately.

For each parallel-for-loop, after generating the cost-expressions, we emit the code shown in Figure 6 that estimates the workload for each iteration of that loop. Here, we use `T` to represent the number of runtime threads (can be obtained by invoking `omp_get_num_threads` function). For each thread, we need to be able to maintain (and retrieve) the cost of each iteration assigned to it. A simplistic design would be that for each thread (with id = `tid`), we maintain (for each thread `tid`) an array of long-integers, such that the $\texttt{i}^{th}$ element will hold the cost of the $\texttt{i}^{th}$ iteration allotted to thread `tid`. This would require an additional loop (iterating over that array) to compute to remaining workload for each thread. For efficiency reasons, we instead store the `TArr[tid]`, as a prefix-sum array, which in turn makes it easier to quickly compute the remaining workload of each thread. Lines 9-10 show the code to maintain the prefix-sum. For example, for a thread with current-iteration-list $\langle t, x, y \rangle$, the remaining workload can be obtained by evaluating $\texttt{TArr}[t][x+y] - \texttt{TArr}[t][x]$. One point to note is that we use the system call `realloc` to allocate the memory (if required) before executing the workload-estimator code.

*Example.* For the code shown in Figure 1, the emitted workload-estimator code can be obtained by replacing the variable `loopBound` with the term `g->N` in Figure 6.

*3.4.2 Runtime Component.* Figure 5b shows the runtime component of our proposed scheme. Before executing any parallel-for-loop, each thread computes the total workload assigned to it at runtime (by invoking the code similar to that shown in Figure 6).

Each OpenMP parallel-for-loop gets translated (by GCC) into three parts (as shown in Figure 2) : (i) assign an initial list of iterations, (ii) execute the assigned iterations, and (iii) obtain next list of iterations to be executed. The last two parts are executed in a loop, till there are iterations to be

```
Function WorkSteal()    // Returns SUCC if it is able to steal any iterations, FAIL otherwise.
2   Lock ( global ) ;
3   v = findVictim() ;
4   if v == -1 then
5       Unlock ( global ) ;
6       return FAIL;
7   busyStatus[v] = true ;
8   Unlock(global) ;                                   } Critical section for all thieves.
9   Lock ( local_v ) ;
10  if (stealPt=WorkDivider(v)) == 0 then
11      Unlock( local_v ) ;
12      return FAIL;
13  Say the CIL of v is ⟨t_j, x, y⟩;
14  Set the CIL of v=⟨t_j, x, stealPt)⟩;
15  Unlock(local_v) ;                                  } Critical section for thief and victim.
16  busyStatus[v] = false ;
17  Set the CIL of thief = ⟨t_j, stealPt+1, y − stealPt⟩ ;
18  return SUCC;
```

Fig. 7. Work-Stealing algorithm. Abbreviations used: CIL = current-iteration-list

```
Function findVictim(ThreadId id):
//Returns victim-id or -1
    victim = -1;
    work_max = 0;
    for i = 0 to T-1 do:
        if busyStatus[i] == true then
            continue
        Say the CIL of t_i = ⟨t_j, x, y⟩;
        work = TArr[j][x + y] - TArr[j][x] ;
        if work > work_max then
            work_max = work;
            victim = i;
    return victim;
```

Fig. 8. WSRW victim finding scheme. $T$ = # threads. Abbreviations used: CIL = current-iteration-list.

executed. Corresponding to these three parts, the runtime component has three sub-components. While the first two sub-components of our design match that of the typical current OpenMP translations, the last sub-component differs, where we invoke our cost ware work-stealing algorithm.

### 3.5 Cost Aware Work Stealing Algorithm

We now discuss our work-stealing algorithm (shown in Figure 7), which implements the work-stealing part of the flow-chart shown in Figure 3. Our algorithm has four interesting properties: (i) the algorithm may be invoked by multiple threads at the same time; (ii) while multiple thieves are trying to identify who will steal first, the victim is not locked; (iii) a victim is chosen based on its remaining workload; and (iv) the algorithm avoids the usage of queues and instead takes advantage of the efficient structure of the current-iteration-list.

The algorithm uses two locks, a global lock which is shared by all the threads, and a local lock which is unique to each victim. The thief takes a global lock before trying to choose the victim and makes sure that only one thief chooses a victim at a time. This ensures that no two thieves should choose the same victim simultaneously. The victim is chosen by invoking the method `findVictim`.

*Algorithm* `findVictim`. The details of this algorithm vary depending on the type of work-stealing scheme: for example, for WSRI, it invokes a code as described in 3.3. For WSRW, Figure 8 shows the code for the `findVictim` method. The code is similar to that of WSRI, except that unlike WSRI (where we choose the victim with maximum number of remaining iterations), in WSRW, we choose the victim with maximum remaining work. To compute the remaining workload for any thread $t_i$, we use the previously populated prefix-sum array `TArr` (Line 18).

In Figure 7, after finding the victim, the thief sets its busyStatus flag and unlocks the global lock. This setting of the status flag ensures that no other thief can choose this victim, till the flag is set to false. After choosing a victim, we take a local lock of that victim thread; each thread has a local lock that is acquired by that thread before it reserves iterations from its own current-iteration-list

```
1  Function WorkDivider(ThiefID v):
2      Say the current-iteration-list of v = ⟨t_j, x, y⟩ ;
3      Threshold = (TArr[x + y] − TArr[x])/2;
4      stealPt = y;
5      for (i = x; i < x + y; i++):
6          cost = TArr[i] − TArr[x];
7          if cost ≥ Threshold then
8              stealPt = i;
9              break;
10     return stealPt;
```

Fig. 9. Work divider for the WSRW scheme.

(Figure 3, box D). We distribute the iterations as per the Work Divider function. Note that in this design, global lock is taken for a very short period, only to find the victim thread (and not for actual stealing, which may take a longer time). We then acquire individual locks on the victim thread's resources, thereby reducing the contention among the thieves to a large extent.

WorkDivider *algorithm.* The details of this algorithm varies based on the strategy of work-stealing. The Algorithm 9 gives details of the WorkDivider function for the WSRW scheme. Ideally our goal is to steal half of the remaining workload of the victim thread. But since we steal in terms of iterations of the loop, we may end up stealing half or less than half of the remaining workload. We identify the index of the earliest iteration (stealPt), such that the iterations of the victim, from $x$ to stealPt, have at least Threshold (set to half the remaining workload) amount of workload. The remaining iterations (from stealPt+1 to $y$) can be stolen; the algorithm returns stealPt. In case of WSRI, the WorkDivider simply returns $y - y/2$ (uses integer divison), if the current-iteration-list of the victim is $\langle t_j, x, y \rangle$.

In Figure 7, if we find that the stealPt (indicating the starting index of the list of iterations to be stolen) is same as $y$ (the last iteration of the victim), then it indicates that there are no iterations to steal. Otherwise, we first update the current-iteration-list of the victim before unlocking $local_v$. Then, we reset the busy flag for the victim (making it available for stealing by other thieves) and set the current-iteration-list of the current thief.

## 4 Optimizations

**Opt1: Redundant Initialization.** The WSRW scheme proposed in Section 3, emits code to populate the TArr array before every parallel-for-loop. Many times it happens that the same parallel-for-loop is run consecutively for many rounds; for example, till a fixed point criteria is reached; for example, in page-rank a parallel-loop is executed till the difference of ranks of pages does not vary 'significantly' between two consecutive rounds. In such cases populating TArr again and again can be redundant. We address this issue by emitting additional code before the parallel-for-loop, to check whether the same parallel-for-loop was executed immediately before. And if so, we don't populate TArr and instead reuse the previously populated values. This optimization can only be applied if the number of iterations of the loop and the variables present in the cost-expression string for the iterations of a loop, do not change between two consecutive invocations of the loop.
**Opt2: Loops with same cost for each iteration.** Our proposed scheme is mainly targeting loops where the cost of the iterations are non-uniform. Thus we identify uniform parallel-for-loops, where all the iterations perform exactly the same amount of work, and then use the default cyclic schedule. This optimization can only be applied if the number of iterations of the loop and the variables present in the cost-expression string for the iterations of a loop, do not change between

```
#pragma omp for
for (i=0;i<N;i++) {
 if (node[i] is marked){process-neighbors.}}
```

Fig. 10. Illustrative pseudo code for a topology driven algorithm.

two consecutive invocations of the loop. This optimization prevents the unnecessary initialization overheads of WSRW where we know we can get better performance using cyclic for such loops.

**Opt3: Loop bodies guarded with a predicate.** A pattern we have found in some kernels is that only a subset of the iterations of the parallel-for-loop perform computation, in each invocation of the parallel-for-loop. For example, Figure 10 shows an illustrative pseudo code showing a similar behavior. Our proposed cost estimation scheme (Section 3.4) computes the cost of an if-else statement as the maximum cost of the then-block and else-block and consequently, may lead to inaccurate work-estimates (especially when the number of active nodes can be a small fraction of the total number of nodes). In such cases, we again fall-back on WSRI owing to its lesser overheads.

**Opt4: Optimized WorkDivider.** The WorkDivider function shown in Figure 7, goes through all the remaining iterations of a thread to identify the value of the variable stealPt. This algorithm is inefficient as the number of iterations per thread can be huge. We exploit the property of `TArr` (a prefix-sum array) and a strategy similar to binary search to devise a more efficient way finding the value of stealPt. Instead of sequentially iterating from $x$ to $x + y$ (in Figure 9), we use the middle point $((2 * x + y)/2)$ to divide the array into two halves. The amount of work in either half can be calculated in $O(1)$ time: The work of left and right halves can be calculated using the expression $\texttt{TArr}[(2*x+y)/2+1] - \texttt{TArr}[x]$ and $\texttt{TArr}[y] - \texttt{TArr}[(2*x+y)/2]$, respectively. If they are exactly dividing the workload into two halves then we set stealPt to the middle point. Otherwise, we repeat this divide and conquer strategy to find the partition such that the iterations of the victim, from $x$ to stealPt, have at least Threshold (set to half the remaining workload) amount of workload. This WorkDivider algorithm is more efficient – complexity is $O(logN)$ (compared to the complexity $O(N)$ for the code shown in Figure 9), where $N$ is the number of iterations of the parallel-loop.

## 5 Discussion

**Reorganise Data vs. Scheduling.** During our study, we observed that not only the scheduling techniques but also the data layout of the input graph can impact the performance of the parallel-for-loops. This is because the data-layout can impact the distribution of the nodes/edges to the threads (and hence load balancing), and the locality of data-access by each thread. We generated multiple versions of the same graphs, where each version has a different organization (for example, vertices sorted by their degree, vertices ordered by their level in the breadth first search (BFS) graphs, and so on). We ran different programs (each program having a fixed schedule) against different versions of the same input graph. While we could not identify any particular reordering scheme as the best scheme across different benchmarks, we still feel that it is an interesting future work to explore the idea of changing the data layout based on the program behavior to improve load imbalance and locality.

**Static Scheduling.** Static scheduling is one of the schedule provided by OpenMP where assignment of loop iterations to thread happens during compile time. This scheme has the least overhead among all the other schedules provided by OpenMP. While it is known that static-scheduling works very well when all the iterations have uniform workload, we have found that for iterations with non-uniform workload, cyclic scheduling (see Section 2)) typically works much better (compared to other scheduling schemes given by OpenMP) when dealing with large real-world scale-free graphs as well as roughly regular graphs. These advantages of cyclic distribution (over static schedule) have also been reported by prior works [18, 32]. The reason why static and the cyclic schedule

work better because OpenMP provides a very highly tuned implementation of these schemes that involve minimal overhead (does not involve locks, or invoke the scheduler, and so on).

An important point to note is that, when the schedule is set to "runtime" and during execution we set the environment variable OMP_NUM_THREADS to "static" (or static,1), then we have found that many times the static schemes (static and cyclic) took more/comparable time compared to guided/dynamic schemes. But when we set the schedule to "static" or "static,1", OpenMP uses a highly efficient scheme to realize these scheduling policies with minimal overheads and that led to efficiency in many of the benchmarks. Thus, we use these efficient numbers of the baseline OpenMP for our comparison.

**Dynamic/Guided Scheduling.** It can be argued that the performance of dynamic and guided scheduling can be improved by tuning the chunk size. However, tuning the chunk size for each application is a difficult and time-consuming task. Therefore, we stick to the default chunk size for our experiments.

We find that in the absence of the right chunk-size for the guided/dynamic schemes, for irregular loops where static/cyclic may not provide the best load-balance, our proposed scheme is most effective.

**Parameters Used In COWS.** There are two parameters that we have used in COWS: (i) the reservation size $c$ (see Figure 3) and (ii) the minimum iteration limit, below which we will not steal iterations from any victim. We ran many experiments and based on that set $c = \sqrt[4]{\text{total-workload}}$ and minimum iteration limit = 5. Note that doing a thorough search of this space to study the behaviour of different formulae and coming up with a scheme to identify the best possible reservation-size and the minimum iteration limit is a challenging task in itself, but is beyond the scope of this paper. We leave it as an interesting future work.

**Identifying the best scheduling policy.** In Section 6, we show that for a variety of kernels our proposed technique performs very well compared to the default scheduling policies of OpenMP, especially in the context of irregular kernels. However, we would like to note that the comparative efficiency of different scheduling policies not only depends on the individual scheduling policies, but also on the specific kernel at hand, and the exact organization of the input. As both of these factors greatly impact the overheads, load-balancing, and even data locality. Consequently, it becomes difficult to identify the best scheduling policy across all the kernels and all their possible inputs. We leave it as a future work to identify the best scheduling policy for any given parallel kernel and input. Additionally, it would be an interesting future work to perform a limit study on how far different scheduling policies are from the optimal possible scheduling for a varied sets of kernels and inputs. This would give directions to the future researchers on where the maximum scope of improvement remains.

**Possible Overheads Of Locking.** In Figure 7, in the first critical section, locking is required only by the thieves (without blocking the workers). In the second critical section, the locking is only between the thief and the worker (note: the overheads of the lock mainly depends only the number of contending threads). These heuristics ensure low overheads, which is also established in our empirical evaluation (Section 6.3).

## 6 Implementation and Evaluation

As discussed in Section 3, our proposed techniques have two components: the compile-time component and the runtime component. We have implemented our proposed compile-time component in IMOP [26], an open-source compiler framework (source-to-source) for OpenMP programs that makes it convenient to write compiler passes. Our implementation spanned $1.3k$ lines of Java code. We have implemented our proposed runtime component in libgomp, a shared library for

OpenMP(Version:4.5) in GCC-11.2; this runtime component spanned 1.4$k$ lines of C code. The runtime component is implemented in such a way that the binary code generated by the existing GCC compiler will use our proposed COWS scheduler based on the value of an environment variable (`OMP_SCHEDULE`). To use any of the discussed schedulers, the program has to set the `schedule` clause in the parallel-for-loop to `runtime` and appropriately set the value of this environment variable – to `WSRW`, `WSRI`, or `WSR`. To perform a comparative evaluation, we also used the schedulers provided by OpenMP (static, cyclic, dynamic, or guided). For dynamic and guided, we use the default chunk size (= 1). We chose a fixed chunk size as it is well known [3] that it is quite challenging to find the right chunk size(s) that will provide the best performance for all the benchmarks. Further, we stick with the default chunk-size for OpenMP, lest we are criticized for having bias in fixing a chunk-size or a set of fixed chunk-sizes. For these three default schedulers, we set the `schedule` clause directly in the for-loop, instead of using the "runtime" schedule option, as the latter was comparatively inefficient than the earlier option. All the input codes were compiled with the –O2 level optimization of GCC.

Our proposed techniques were evaluated on seven popular shared-memory kernels shown in Figure 11: K-committee (KC) – creates committees such that no two members in the committee are at a distance > a pre-defined value $K$; leader election for general graphs (LE) – finds a leader in a parallel/distributed environment; breadth first search (BFS) – computes the diameter of a rooted graph; bellman ford (BF) – computes distance of every node from the root; dominating set (DS) – finds dominating set of the given graph; page rank (PR) – iteratively computes the rank of each node in the graph; and connected components (CC) – computes the number of connected components in the graph; The first five kernels are taken from IMSuite [13] (we excluded the remaining kernels implementing randomised algorithms, as their base execution times varied significantly) and remaining two were written based on kernels used in prior literature [30]. Figure 11 also lists some characteristics of these kernels. These include, number of lines of code, number of static parallel-for-loops, the number of dynamic parallel-for-loops encountered in the context of the five above discussed inputs, and the different optimizations (discussed in Section 4) that were applicable for the benchmark. We see that in case of KC, the number of dynamic parallel-for-loops are same across the inputs. This is because, the number of parallel-for-loops in KC depend only on the value of $K$ (maximum distance among the members of the community) – a constant value, irrespective of the input.

To generate the cost of input independent statements, the profiling was done using a very small input (randomly generated graph of 32 nodes). We observed that the total compilation time overheads were negligible.

All these kernels have a common characteristic that they have at least one irregular loop. Given skewed inputs (for example, power-law graphs) these codes lead to highly non-uniform workloads, with high amount of imbalance among the iterations of the parallel-for-loops. Since the main target of our proposed scheme is to realize high performance in codes and inputs leading to non-uniform workloads, we will first focus on such workloads. To do so, we picked three input graphs from SNAP (i.e. Stanford Large Network Dataset Collection) [23]: Youtube, LiveJournal and Orkut graphs. Besides testing the efficacy of our proposed system in the context of highly non-uniform workloads, we also tested our code against inputs where the resulting workload among the iterations of the parallel-for-loops is more uniform – this works as the worst-case scenario for us where the overheads due to our proposed scheme are most striking. To do so, we picked two roughly regular input graphs from SNAP: RoadNet-CA and RoadNet-PA. Figure 12 shows some characteristics of all these datasets.

We performed the evaluation on two systems: (i) Aqua: a node in a high-end super-computing cluster containing an Intel(R) Xeon(R) Gold 6248 processor, with two sockets, each having 20 cores

| Bench | LOC | SL | UL | DA | DB | DC | DD | DE | Optimizations Applied | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | Opt1 | Opt2 | Opt3 | Opt4 |
| 1. KC | 403 | 6 | 2 | 28 | 28 | 28 | 28 | 28 | F | T | F | T |
| 2. LE | 269 | 3 | 2 | 34 | 30 | 16 | 1288 | 1044 | T | T | F | T |
| 3. BFS | 231 | 2 | 1 | 15 | 15 | 9 | 556 | 543 | T | T | T | T |
| 4. BF | 258 | 3 | 1 | 40 | 40 | 25 | 1666 | 1624 | F | T | F | T |
| 5. DS | 525 | 12 | 4 | 949 | NA | NA | 212 | 157 | F | T | F | T |
| 6. PR | 184 | 2 | 1 | 30 | 26 | 52 | 44 | 44 | T | T | F | T |
| 7. CC | 170 | 2 | 1 | 7 | 9 | 4 | 243 | 249 | F | F | F | T |

Fig. 11. Benchmarks used for evaluations and their characteristics. Abbreviations: LOC: Lines of code; SL: #static parallel-for-loops; UL: #uniform parallel-for-loops; DA, DB, DC, DD and DE are the #dynamic parallel-for-loops for Youtube, LiveJournal, Orkut, RoadNet-CA and RoadNet-PA respectively; Opt1, Opt2, Opt3 and Opt4 are the four optimizations discussed in Section 4.

| Input | #Vertices | #Edges | Min degree | Max degree |
|---|---|---|---|---|
| Youtube | 1.13M | 2.98M | 1 | 28.8K |
| LiveJournal | 4.84M | 68.99M | 1 | 2.7K |
| Orkut | 3.07M | 117.18M | 1 | 27.5K |
| RoadNet-CA | 1.96M | 2.76M | 1 | 12 |
| RoadNet-PA | 1.08M | 1.54M | 1 | 9 |

Fig. 12. Characteristics of the datasets used.

(total 40 cores) and total memory of 192GB. and (ii) Goodwill: a standalone server containing an Intel(R) Xeon(R) Gold 6240 processor, with two sockets, each having 18 cores (total 36 cores) with SMT enabled; leading to a maximum of 72 hardware threads, and total memory of 125GB.

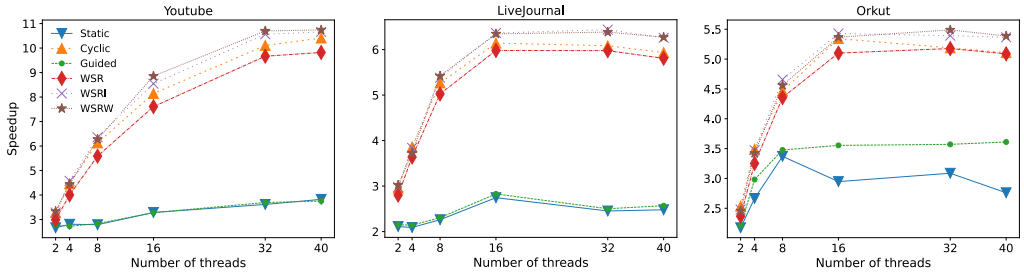
For each benchmark kernel, we evaluated seven different schemes of scheduling: i) `dynamic` ii) `static` iii) `cyclic` iv) `guided` v) WSR vi) WSRI vii) WSRW. We studied the performance of these scheduling schemes for varying number of hardware threads (in powers of two). In specific, we pay special attention to the cyclic schedule, which we found (in our experiments) to be the best among the default schedules supported by OpenMP, in majority of the cases. For running a program on a system with $T$ number of hardware threads, we set the number of software threads (`OMP_NUM_THREADS`) also to $T$.

To understand the impact of our proposed techniques we show the evaluation in five dimensions: (i) performance gains in the context of highly irregular graphs, (ii) behavior in the context of roughly regular graphs, (iii) overheads for victim selection, (iv) a study on the number of steals operations, and (v) impact of proposed optimizations.

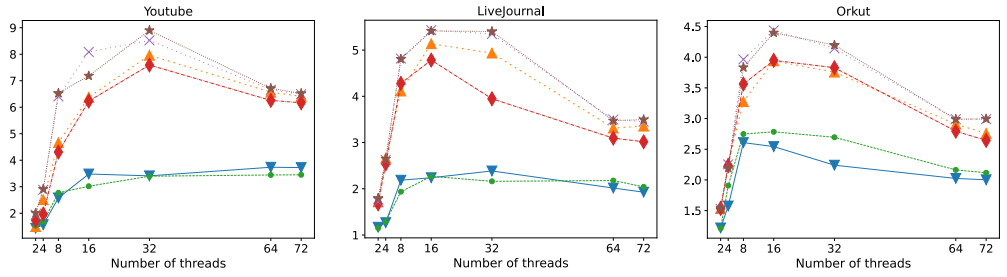
### 6.1 Performance Gains in Highly Irregular Graphs

In this section, the graphs show the speedups obtained due to various scheduling schemes, compared to the OpenMP dynamic schedule, for the three power-law graph inputs. Among the different default scheduling schemes of OpenMP, the best scheduling scheme varies depending on the specific benchmark, input and number of runtime-threads. We have found that cyclic schedule performed the best among the default scheduling schemes for the most part. Thus, for each kernel, we additionally highlight the performance of WSRI and WSRW compared to the cyclic schedule (as a geomean average).

**KC.** Figure 13 shows the speedups obtained for the KC kernel. We compute the speedup of a kernel due to a schedule X, for a given configuration (input dataset, System and the number of hardware threads), using the formula: (time taken to execute the kernel using dynamic-schedule)/(time taken to execute the kernel using schedule X).




(a) System = Aqua




(b) System = Goodwill

Fig. 13. Kernel = KC. Speedups with respect to dynamic scheduling. The points have been connected only to improve the visibility.

Our schemes WSRW and WSRI, clearly outperform all the other schemes. Overall, we see that on the Aqua system the speedups varied between 2.47× to 10.741× for WSRW and 2.479× to 10.66× for WSRI. Similarly, on the Goodwill system the speedups varied between 1.538× to 8.899× for WSRW and 1.529× to 8.522× for WSRI.

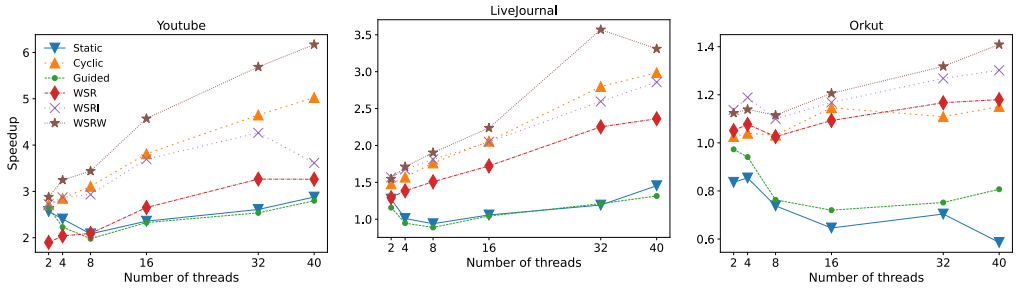
It can be seen that the performance of WSRW is comparable to WSRI. It is because, here, two parallel-for-loops that are the primary source of workload had while-loops because of which we WSRW falls back to WSRI (see Section 3.4.1).

Compared to the cyclic-schedule, for varying number of cores and across all the three inputs, WSRW (and WSRI) was 1.10× and 1.03× better on Goodwill and Aqua system, respectively.

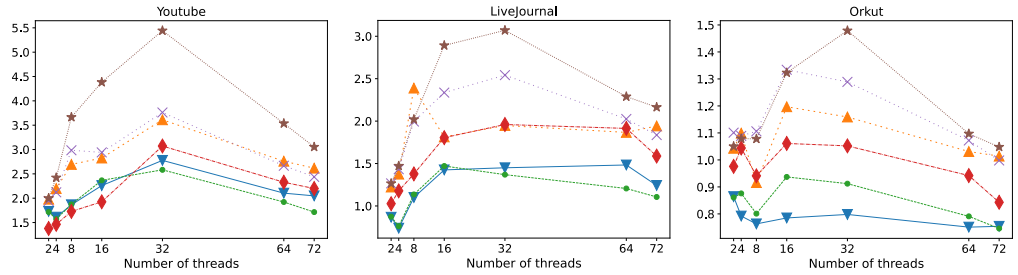
On the Aqua system, when the number of threads increased from 16 to 32, performance does not scale at the same (high) rate. This is because, it is a two-socket machine and the inter-socket-communication overheads start when the number of threads are more than 20 cores (as each socket on Aqua has 20 cores). A similar trend can be seen on Goodwill, where each socket has 18 cores. Secondly, there is a dip in performance gains when we increase the threads from 32 to 64 or 72. This is because after when we increase the number hardware threads to beyond 36, we start running our cores on the SMT threads, where the performance does not scale up like the regular cores. We have observed similar hardware based behaviours on the other four kernels as well.

**LE.** Figure 14 shows the speedups obtained for the LE kernel. Overall we see that for the LE kernel, WSRW clearly outperforms all the other schemes. Overall we see that on the Aqua system the speedups varied between 1.115× to 6.171× for WSRW and 1.1× to 4.265× for WSRI. Similarly, on the Goodwill system the speedups varied between 1.048× to 5.443× for WSRW and 0.998× to 3.761× for WSRI.

It was seen that performances of both WSRI and cyclic was comparable on Aqua but on Goodwill, WSRI performed better. We attribute this to the faster memory subsystem on Aqua leading to




(a) System = Aqua




(b) System = Goodwill

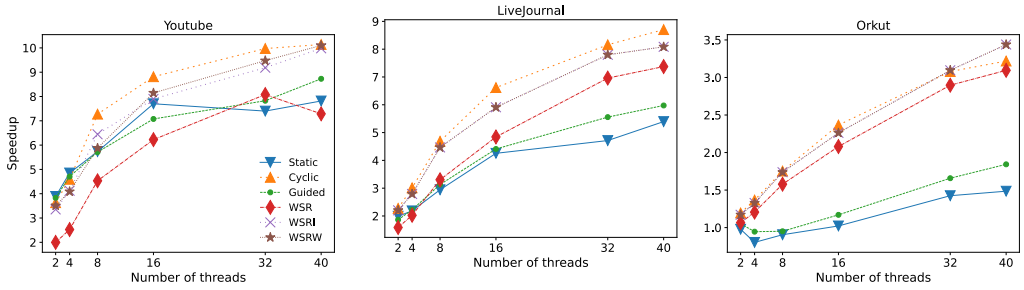
Fig. 14. Kernel = LE. Speedups with respect to dynamic scheduling. The points have been connected only to improve the visibility.

lesser overheads. On LE, we can observe an approximate order among the different scheduling schemes: WSRW, WSRI, cyclic, WSR, guided, static, dynamic. The only exception was in the Orkut input, where we see that static and guided performed worse than dynamic. We have found similar degradation in static and guided in other kernels as well (PR, DS, and CC). The reason for the degradation in static schedule is the high load imbalance due to the naive scheduling policy. The guided scheme suffered because of (i) the inability to reallocate after allocation, (ii) high contention for locking.

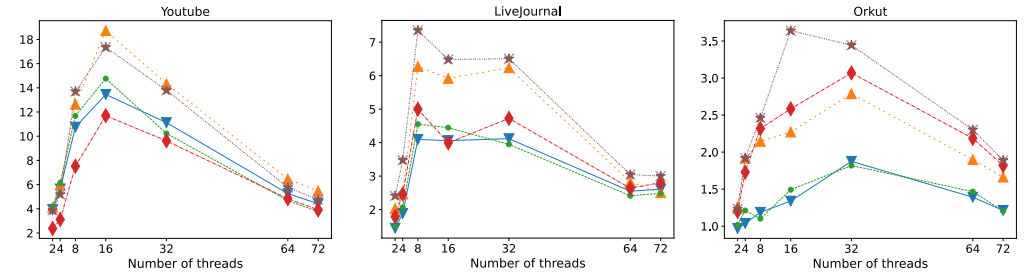
Compared to the cyclic-schedule, for varying number of cores and across all the three inputs, WSRW was 1.13× and 1.18× better on Aqua and Goodwill system, respectively. Similarly, compared to the cyclic-schedule, WSRI was 1.01× and 1.05× better on Aqua and Goodwill system, respectively.

**BFS.** Figure 15 shows the speedups obtained for the BFS kernel. Overall we see that for the BFS kernel, WSRW, WSRI and cyclic outperform the rest in majority of the cases. Also, it is difficult to find a clear winner among these three for BFS on these inputs. Overall we see that on the Aqua system the speedups varied between 1.17× to 9.98× for WSRW (and WSRI). Similarly, on the Goodwill system the speedups varied between 1.233× to 17.352× for WSRW (and WSRI). The performance of WSRW and WSRI look similar due the application of Opt3 (Section 4) on the main loop.

Compared to the cyclic schedule, for varying number of cores and across all the three inputs, WSRW (and WSRI) was 0.95× and 1.09× better on Aqua and Goodwill system, respectively. The reason for this slightly lower performance of WSRW and WSRI, compared to cyclic in the context of BFS is that in this kernel, though WSRI and WSRW pay the respective overheads, the work estimate is still not accurate. Thus we do not get much gains compared to cyclic (a highly efficient




(a) System = Aqua




(b) System = Goodwill

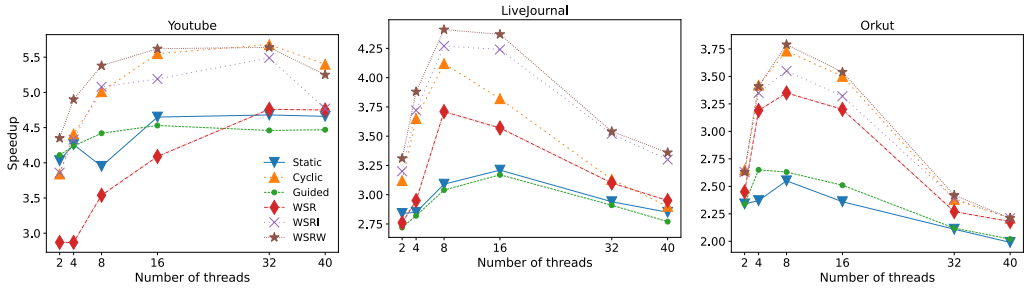
Fig. 15. Kernel = BFS. Speedups with respect to dynamic scheduling. The points have been connected only to improve the visibility.

scheme of scheduling with minimal overheads; see Section 5). However, WSRW and WSRI perform better than the remaining scheduling schemes.

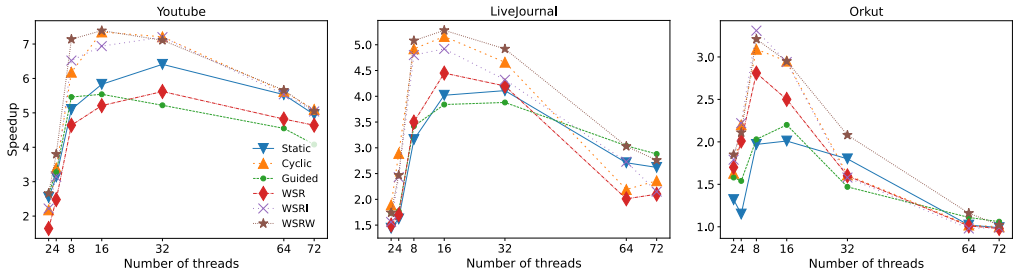
**BF.** Figure 16 shows the speedups obtained for the BF kernel. Overall we see that for the BF kernel, WSRW outperform all the other schemes, except for 72 threads on Goodwill for Orkut and LiveJournal inputs, where the guided scheme narrowly outperforms WSRW. We believe that some low level issues arising out of enabled SMTs at 72 threads is leading to the relatively improved performance of the guided schedule. WSRI and cyclic outperform each other at many points. Overall we see that on the Aqua system the speedups varied between 2.21× to 5.64× for WSRW and 2.21× to 5.49× for WSRI. Similarly, on the Goodwill system the speedups varied between 1.02× to 7.39× for WSRW and 0.98× to 7.2× for WSRI.

Compared to the cyclic-schedule, for varying number of cores and across all the three inputs, WSRW was 1.06× and 1.05× better on Aqua and Goodwill system, respectively. Similarly, compared to the cyclic-schedule, WSRI was 0.98× and 1.0× better on Aqua and Goodwill system, respectively.

**DS.** Figure 17 shows the speedups obtained for the DS kernel. The base kernel ran out of memory for Orkut (on both the systems), and for LiveJournal the was taking many hours to complete a single run. Hence, we have limited our evaluation to Youtube (on Aqua and Goodwill) and LiveJournal (on Aqua) only. Overall we see that for the DS kernel, WSRI outperforms WSRW in majority of the cases and WSRW is the second best schedule. There are 12 parallel for loops in this kernel, hence the overhead of initialization of TArr does slightly impact the performance of the WSRW. Overall we see that on the Aqua system the speedups varied between 1.05× to 1.28× for WSRW and 1.02× to 1.31× for WSRI. Similarly, on the Goodwill system the speedups varied between 1.11× to 1.64× for WSRW and 1.2× to 1.77× for WSRI.

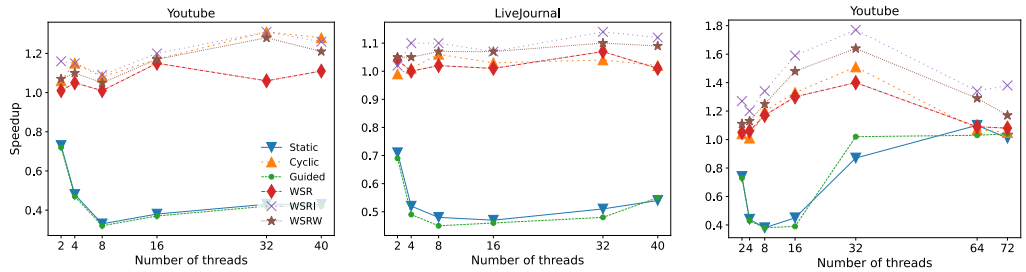



(a) System = Aqua




(b) System = Goodwill

Fig. 16. Kernel = BF. Speedups with respect to dynamic scheduling. The points have been connected only to improve the visibility.



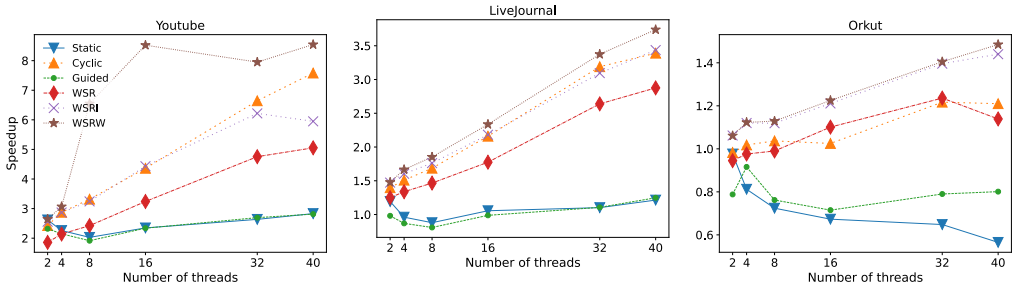

(a) System = Aqua

(b) System = Goodwill

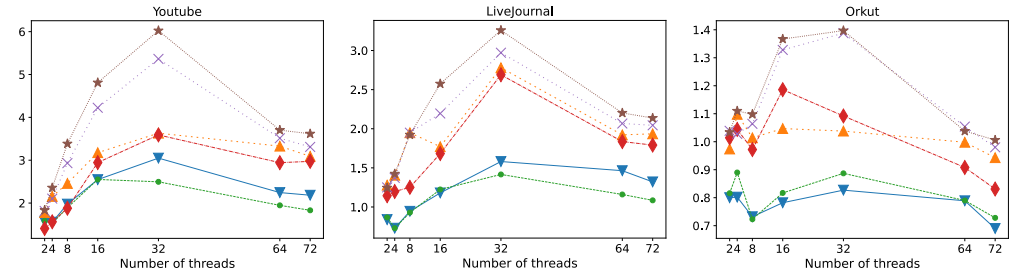
Fig. 17. Kernel = DS. Speedups with respect to dynamic scheduling. The points have been connected only to improve the visibility.

Compared to the cyclic-schedule, for varying number of cores and across all the three inputs, WSRW was 1.01× and 1.10× better on Aqua and Goodwill system, respectively. Similarly, compared to the cyclic-schedule, WSRI was 1.04× and 1.20× better on Aqua and Goodwill system, respectively.

**PR.** Figure 18 shows the speedups obtained for the PR kernel. Overall we see that for the PR kernel, WSRW and (to a large extent) WSRI clearly outperform all the other schemes. Overall we see that on the Aqua system the speedups varied between 1.085× to 8.103× for WSRW and 1.062× to 6.218× for WSRI. Similarly, on the Goodwill system the speedups varied between 1.006× to 6.024× for WSRW and 0.98× to 5.364× for WSRI.




(a) System = Aqua




(b) System = Goodwill

Fig. 18. Kernel = PR. Speedups with respect to dynamic scheduling. The points have been connected only to improve the visibility.

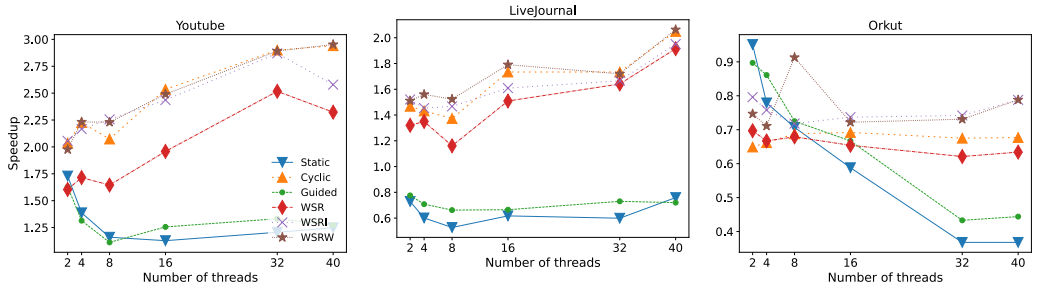
Compared to the cyclic-schedule, for varying number of cores and across all the three inputs, WSRW was 1.15× and 1.16× better on Aqua and Goodwill system, respectively. Similarly, compared to the cyclic-schedule, WSRI was 1.03× and 1.10× better on Aqua and Goodwill system, respectively.

**CC.** Figure 19 shows the speedups obtained for the CC kernel. Overall we see that for the CC kernel, even though WSRW mostly outperform WSRI, WSR, cyclic, static, and guided, WSRW, WSRI, and cyclic had comparable performance at many points. Overall we see that on the Aqua system the speedups varied between 0.711× to 2.952× for WSRW and 0.718× to 2.87× for WSRI. Similarly, on the Goodwill system the speedups varied between 0.703× to 3.279× for WSRW and 0.671× to 2.867× for WSRI.

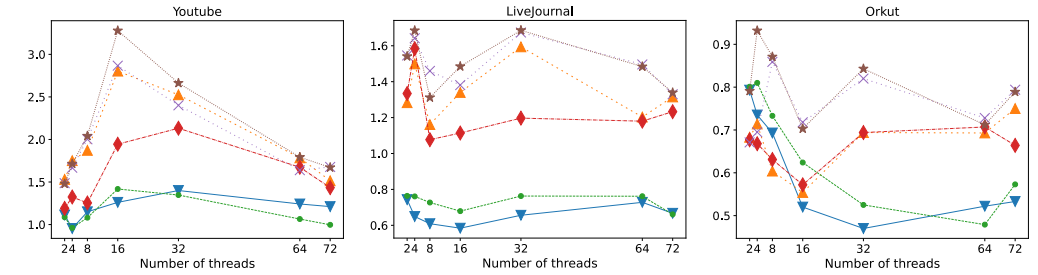
The Orkut input data-set shows an interesting point: the dynamic schedule out-performs the rest. We believe this particular behavior is because of the way in which the data is organized in the Orkut dataset.

Compared to the cyclic-schedule, for varying number of cores and across all the three inputs, WSRW was 1.06× and 1.12× better on Aqua and Goodwill system, respectively. Similarly, compared to the cyclic-schedule, WSRI was 1.03× and 1.08× better on Aqua and Goodwill system, respectively.

**Summary.** Across five different benchmark programs, using the three different power-law input datasets, on two different hardware, across varying number of threads we show that WSRW and WSRI get on average (geomean) performance gains of 1.10× and 1.05×, respectively (max 2× and 1.57×, respectively) over the cyclic schedule (the overall best among the default schedules provided by OpenMP). Further, considering the fact that these gains are obtained on top of GCC -O2 level optimized codes, we argue that these gains are significant.




(a) System = Aqua




(b) System = Goodwill

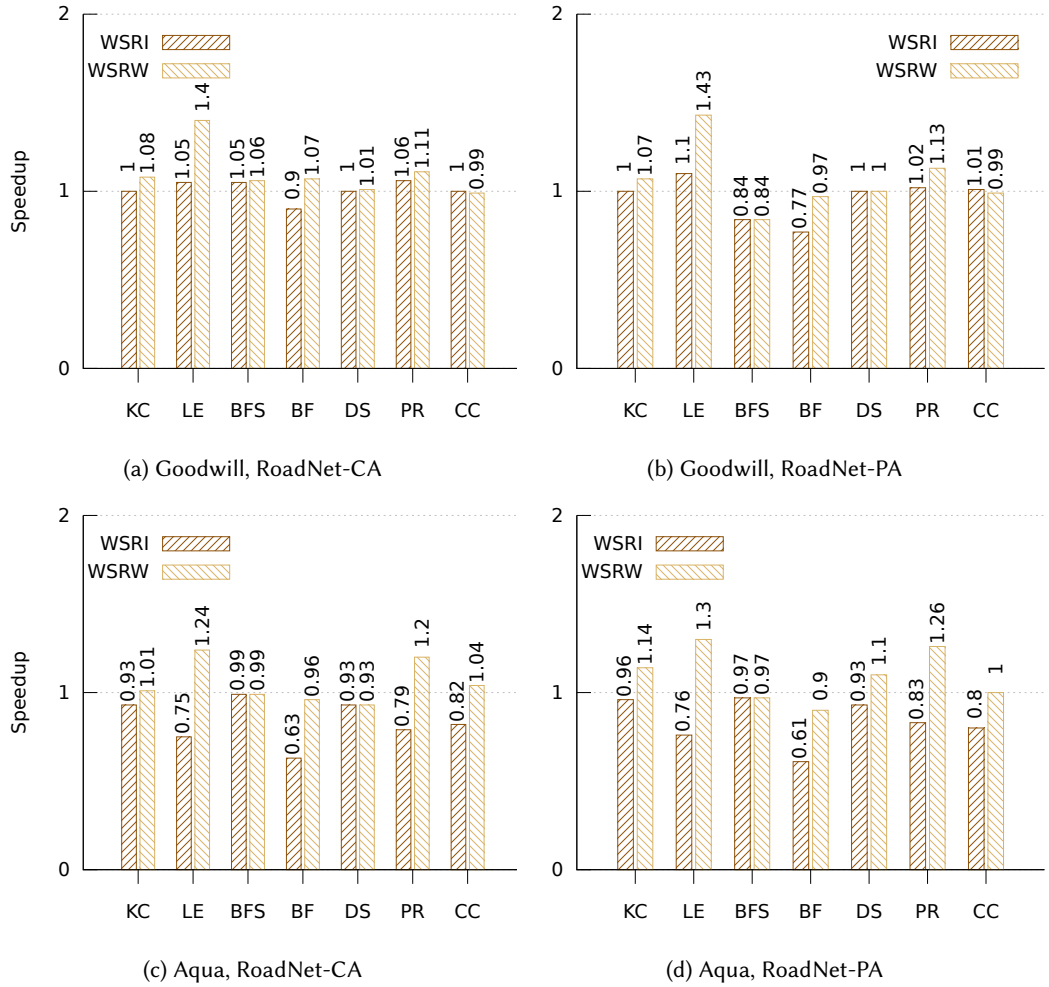
Fig. 19. Kernel = CC. Speedups with respect to dynamic scheduling. The points have been connected only to improve the visibility.

### 6.2 Behavior in the Context of Roughly Regular Graphs

The main target of our proposed scheduling scheme are the graphs that lead to high imbalance among the iterations of parallel-for-loops. To understand the possible negative impact of our proposed scheduling schemes, we evaluated them using the same set of kernels (Figure 11) on roughly regular graph datasets (RoadNet-CA and RoadNet-PA, shown in Figure 12). For brevity, we only show the performance results for the max number of cores available on the respective hardware systems; we have observed that the numbers for the remaining cores are similar.

The Figure 20 shows the performance of WSRW and WSRI shown as speedups over cyclic-scheduling - the overall best among the default OpenMP scheduling techniques; for brevity, we skip the comparison with other schedules. Overall, we can see that WSRW performs slightly better than cyclic-scheduling, while WSRI performs slightly worse. Note that though these inputs are roughly regular, the iterations of the parallel-loops do not have have the exact same workload, thus leading to imbalance (albeit less compared to the power-law graphs) which can be exploited by COWS. WSRW shows gains where the overheads of WSRW are compensated by the gains resulting from improved load-balance. Compared to WSRI, we believe that WSRW performs better because it uses profiling information, whereas WSRI does not.

Across the two datasets and both the hardware, we find that WSRW obtains speedups of 1.07×, 1.34×, 0.96×, 0.97 ×, 1.01×, 1.17× and 1×, for KC, LE, BFS, BF, DS, PR and CC respectively. Similarly, across the two datasets and both the hardware, we find that WSRI obtains speedups of 0.97×, 0.9×, 0.96×, 0.73 × ,0.97×, 0.92× and 0.90×, for KC, LE, BFS, BF, DS, PR and CC respectively. We see that even in the context of roughly regular graphs, WSRW performs reasonably well, without much impact due to its overheads.




(a) Goodwill, RoadNet-CA (b) Goodwill, RoadNet-PA

(c) Aqua, RoadNet-CA (d) Aqua, RoadNet-PA

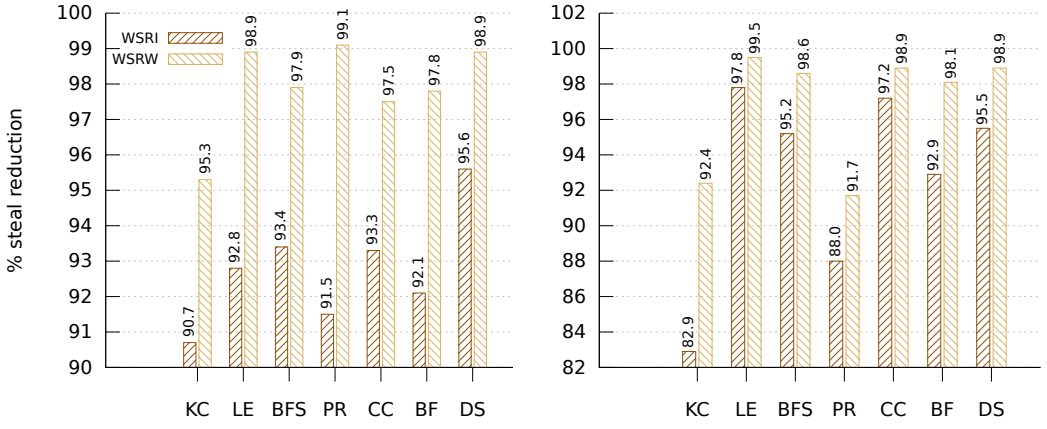
Fig. 20. Speedup of WSRW and WSRI over cyclic-scheduling on roughly regular graphs. Number of threads = maximum # cores.

| Overheads | KC | LE | BFS | BF | DC | PR | CC | Avg |
|---|---|---|---|---|---|---|---|---|
| i. | 0.02% | 0.025% | 0.14% | 0.01% | 0.01% | 0.45% | 2.4% | 0.43% |
| ii. | 0.007% | 0.001% | 0.05% | 0.03% | 0.02% | 0.001% | 0.05% | 0.03% |
| total | 0.027% | 0.026% | 0.19% | 0.04% | 0.03% | 0.451% | 2.45% | 0.47% |

Fig. 21. Overheads for victim selection.

### 6.3 Overheads for victim selection

There are two main components of overheads for victim selection (i) Initialization cost of TArr array. (ii) Cost paid by a thief to find the victim (may not be part of the critical-path). We have found that the total overhead is a small percentage of the execution time. For example, Figure 21 shows the overheads for victim selection for different benchmarks on the Aqua system. It can be




(a) Aqua (b) Goodwill

Fig. 22. % reduction in the number of steal operations for WSRI and WSRW schemes, compared to WSR.

seen that on average the overhead was very low (0.47%). This attests to the efficacy of our proposed optimization. The results were similarly low for Goodwill; numbers omitted for brevity.

### 6.4 A study on the number steal operations

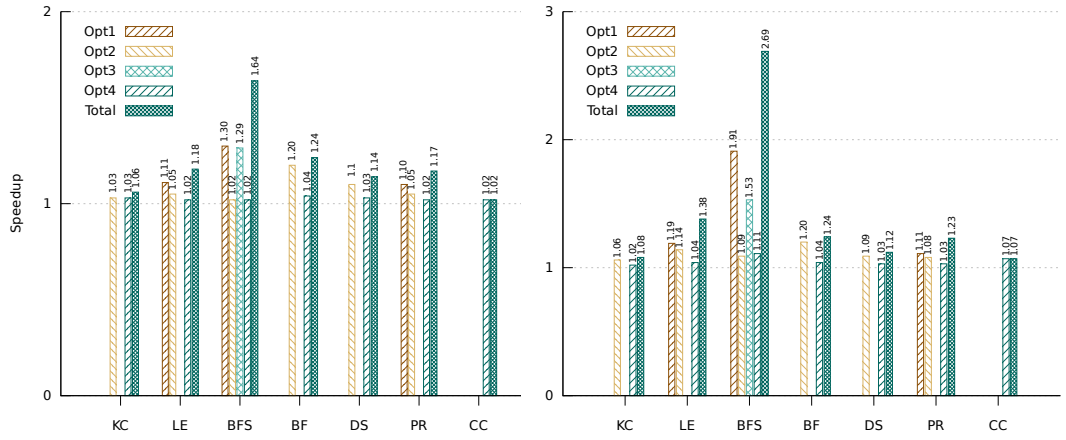
We also compared the number of successful steal operations performed by different work-stealing schedulers (WSR, WSRI and WSRW). Figure 22 shows reduction in the number of steal operations, for WSRI and WSRW compared to WSR. Though this reduction does not directly correspond to increase in performance but the improvement does indicate that the load balancing is likely to be better. Further, we can see that WSRW has significantly lower number of steal operations compared to WSRI. This is due to the fact that WSRW steals based on the amount of remaining workload and not based on the remaining number of iterations, this in turn leads to better load balance in WSRW, which is reflected in the fewer number of steals.

The findings are inline with the common understanding that neither too many, nor too few steals may be beneficial. For example, the classical scheduling schemes do not perform any steals, but still perform worse, as they totally lose out on the benefits of work-stealing.

### 6.5 Impact of Optimizations

To study the impact of the proposed optimizations (Section 4) we have compared the performance of WSRW, with and without the proposed optimizations. Figure 23 shows the geometric mean performance gains (across all the five datasets) due to each of our proposed optimizations; when the kernels are run using the maximum number of available hardware cores on each system. Recall from Figure 11 that not all optimizations were applicable to all the benchmarks, and hence for each benchmark, Figure 23 shows only the applicable bars for the optimizations. Opt1 and Opt3 provide the most significant performance boost wherever applicable. Impact of various optimizations ranges from 1.02× to 1.45× on Aqua and 1.01× to 1.33× on Goodwill.

**Overall Summary of Evaluation.** Overall, we find that compared to the default scheduling schemes supported by OpenMP, our proposed scheduling technique leads to clear gains in the context of parallel-for-loops with high workload imbalance among its iterations, while giving us acceptable performance in the context of parallel-for-loops with more or less balanced workloads among the iterations. Our evaluation over seven different benchmark programs, using five different input datasets, on two different hardware across a varying number of threads (leading to a total of 275 number of configurations) shows that in 225 out of 275 configurations, compared to the




(a) Aqua (b) Goodwill

Fig. 23. Impact of various optimization on WSRW. Speedups baseline: WSRW with no optimizations.

best OpenMP scheduling scheme for that configuration, WSRW achieves clear performance gains (maximum upto 2 ×). In the remaining 50, except for one configuration where the degradation was 30% (Figure. 19b, Orkut graph, 16 threads), the worst case degradation was not more than 10%. Further, unlike in the default schemes of OpenMP, we do not need to manually choose a chunk-size in COWS and still get high performance. We also show that the proposed optimizations are impactful.

## 7 Related Work

**Loop-scheduling.** There have been many works that go beyond the default set of scheduling schemes (static, dynamic, guided) supported by OpenMP [17, 21, 24, 27]. Especially considering the challenges in scheduling irregular parallel loops, there have been many focused attempts in this direction. Many work-sharing scheduling strategies have been implemented in LLVM. [19] shows that none of the scheduling strategies of LLVM were significantly superior to the rest across all the benchmarks. And most of the time their scheme were similar to that of dynamic/guided. In contrast, as shown in Section 6, our proposed scheme outperforms (upto 4 ×) these two in all the seven benchmark kernels, for all inputs, for varying number of threads (except for CC kernel, and Orkut input). BinLPT [28] is a workload-aware loop scheduler that takes the estimate of the workload of the parallel-for-loop from the user to distribute the iterations. Thoman et al. [35] also present a scheme that uses a combined compiler and runtime approach where they compute the 'effort estimation functions' for the parallel loops. One main issue with their technique is that their scheme mainly works on affine loops with no heap access. Prabhu and Nandivada [29] extend this idea to handle arbitrary loops. One main issue with all these schemes is that they assign the iterations of the loop before the loop starts executing and do not have provisions for dynamically re-assigning the iterations based on the actual execution. In contrast, our scheme does dynamic load balancing by performing work-stealing based on the estimate of the remaining workload. It would be interesting to extend our work with some of the prior techniques of loop scheduling to improve processor affinity and cache locality [9, 10, 15, 16, 34].

**Profile-guided Optimization.** Profile-guided optimizations have been used for a long time to improve the performance of parallel programs. For example, Chen et al. [8] use profiling to improve the mapping of processes to different processors and Shrivastava and Nandivada [33] use profiling to migrate the threads and scale the frequency of processors for saving energy. Similarly,

Kejariwal et al. [20] and Bull [4] use profiling (history) information to efficiently chunk future instances of parallel-for-loops. In this paper, we use the profile-guided information to compute the cost of parallel-for-loops, which helps in efficiently choosing the victim for work-stealing. This overall improves the performance of our scheme.

**Work-stealing.** Another popular approach to balance the non-uniform workload of the parallel-for-loop is work-stealing [1, 2, 12, 14]. Prior works [31, 36] have shown that even though Cilk supports work-stealing for task-parallelism, for data-parallel loops OpenMP outperforms Cilk significantly. Durand et al. [11] show that stealing half the iterations of the victim is an effective strategy. We improvise on this by stealing iterations which will reduce the workload of the victim by half. We shows that our proposed scheme of cost aware work stealing outperforms the baseline OpenMP. Recently Booth and Lane [3] present a work-stealing with dynamic chunk size based scheduling (victim randomly chosen) that uses the history of the loop-execution to identify the "reservation size" for each thread. They only compare their scheme against dynamic/guided (and not against cyclic – one of the best performing schemes of OpenMP) and show that their performance is comparable to guided/dynamic. In contrast, we propose a cost-aware work stealing scheme, which uses the remaining-workload to identify the victim and the reservation size. Further, our scheme has very low overheads and overall performs better than almost all of the schedules supported by OpenMP (including cyclic).

There have been prior works [11, 22] that perform work-stealing NUMA aware work-stealing. We believe that our proposed techniques can be extended with those ideas to realize improved performance in NUMA aware systems.

## 8 Conclusion

In this paper we presented a scheme that efficiently extends the idea of work-stealing to OpenMP, by taking into consideration the remaining workload with each thread; we call our scheme COst aware Work Stealing (COWS, in short). We present two variations of COWS: WSRI (based on remaining number of iterations) and WSRW (based on the amount of remaining workload). Internally COWS uses an efficient representation of assigned work and performs work-stealing with minimal overheads. We evaluated our implementation on seven different benchmark kernels, using five different input datasets, on two different hardware, for varying number of threads (leading to a total 275 configurations) and show that in 225 out of 275 configurations, compared to the best OpenMP scheduling scheme for that configuration, our approach achieves clear performance gains. We argue that COWS gives the best of both worlds: the low overheads of static scheduling and dynamic load balancing of work-stealing.

**Acknowledgements**: This work is partially supported by SERB CRG grant (sanction number CRG/2022/006971) and NSM research grant (sanction number MeitY/R&D/HPC/2(1)/2014).

## References

[1] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.* 30, 8 (aug 1995), 207–216.

[2] Robert D. Blumofe and Dionisios Papadopoulos. 1998. *Hood: A User-Level Threads Library for Multiprogrammed Multiprocessors.* Technical Report.

[3] Joshua Dennis Booth and Phillip Allen Lane. 2022. An Adaptive Self-scheduling Loop Scheduler. *CCPE* 34, 6 (2022).

[4] J. Mark Bull. 1998. Feedback Guided Dynamic Loop Scheduling: Algorithms and Experiments. In *EuroPar*. 377–382.

[5] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. ACM, 51–61. https://doi.org/10.1145/2093157.2093165

[6] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *IJHPCA* 21, 3 (2007), 291–312. https://doi.org/10.1177/1094342007078442

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. *SIGPLAN Not.* 40, 10 (oct 2005), 519–538. https://doi.org/10.1145/1103845.1094852

[8] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. 2006. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *ICS*.

[9] Q. Chen, M. Guo, and Z. Huang. 2012. CATS: Cache Aware Task-stealing Based on Online Profiling in Multi-socket Multi-core Architectures. In *ICS*.

[10] Alejandro Duran, Julita Corbalan, and Eduard Ayguade. 2008. An adaptive cut-off for task parallelism. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 1–11. https://doi.org/10.1109/SC.2008.5213927

[11] Marie Durand, François Broquedis, Thierry Gautier, and Bruno Raffin. 2013. An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines, Vol. 8122. https://doi.org/10.1007/978-3-642-40698-0_11

[12] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. *SIGPLAN Not.* 33, 5 (may 1998), 212–223. https://doi.org/10.1145/277652.277725

[13] Suyash Gupta and V. Krishna Nandivada. 2015. IMSuite: A benchmark suite for simulating distributed algorithms. *JPDC* 75, 0 (2015), 1–19.

[14] Robert H. Halstead. 1985. MULTILISP: A Language for Concurrent Symbolic Computation. *TOPLAS* 7, 4 (oct 1985), 501–538. https://doi.org/10.1145/4472.4478

[15] B. Hamidzadeh, L. Y. Kit, and D. J. Lilja. 2000. Dynamic Task Scheduling Using Online Optimization. *IEEE Trans. Parallel Distrib. Syst.* 11, 11 (2000), 1151–1163. https://doi.org/10.1109/71.888636

[16] B. Hamidzadeh and D. J. Lilja. 1994. Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing. In *ICPP*. 39–46.

[17] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. 1991. Factoring: A Practical and Robust Method for Scheduling Parallel Loops. In *Proceedings of SC*. ACM, 610–632.

[18] IBM Corporation. 2021. Distribution Techniques. https://www.ibm.com/docs/en/pessl/5.3.0?topic=distributions-distribution-techniques

[19] Franziska Kasielke, Ronny Tschüter, Christian Iwainsky, Markus Velten, Florina M. Ciorba, and Ioana Banicescu. 2019. Exploring Loop Scheduling Enhancements in OpenMP: An LLVM Case Study. In *ISPDC*. 131–138.

[20] Arun Kejariwal, Alexandru Nicolau, and Constantine D. Polychronopoulos. 2006. History-aware Self-Scheduling. In *2006 International Conference on Parallel Processing (ICPP'06)*. 185–192. https://doi.org/10.1109/ICPP.2006.49

[21] C.P. Kruskal and A. Weiss. 1985. Allocating Independent Subtasks on Parallel Processors. *IEEE TOSEM* SE-11, 10 (1985), 1001–1016. https://doi.org/10.1109/TSE.1985.231547

[22] Vivek Kumar. 2020. PufferFish: NUMA-Aware Work-stealing Library using Elastic Tasks. In *HiPC*. 251–260.

[23] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[24] Steven Lucco. 1992. A Dynamic Scheduling Method for Irregular Parallel Programs. *SIGPLAN Not.* 27, 7 (jul 1992), 200–211. https://doi.org/10.1145/143103.143134

[25] V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. 2013. A Transformation Framework for Optimizing Task-Parallel Programs. *TOPLAS* 35, 1, Article 3 (apr 2013), 48 pages. https://doi.org/10.1145/2450136.2450138

[26] A Nougrahiya and V. Krishna Nandivada. 2018. IIT Madras OpenMP (IMOP) Framework.

[27] OpenMP Architecture Review Board. 2021. OpenMP Application Program Interface Version 5.2. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[28] P Penna, A Gomes, M Castro, P.D.M. Plentz, H Freitas, F Broquedis, and J-F Méhaut. 2019. A Comprehensive Performance Evaluation of the BinLPT Workload-Aware Loop Scheduler. *CCPE* 31, 18 (2019), 1–22.

[29] Indu K. Prabhu and V. Krishna Nandivada. 2020. Chunking Loops with Non-Uniform Workloads. In *ICS* (Barcelona, Spain). ACM, Article 40, 12 pages. https://doi.org/10.1145/3392717.3392763

[30] Anchu Rajendran and V. Krishna Nandivada. 2020. DisGCo: A Compiler for Distributed Graph Analytics. *TACO* 17, 4, Article 28 (sep 2020), 26 pages. https://doi.org/10.1145/3414469

[31] Solmaz Salehian, Jiawen Liu, and Yonghong Yan. 2017. Comparison of Threading Programming Models. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 766–774.

[32] Jun Shirako, Jisheng M. Zhao, V. Krishna Nandivada, and Vivek N. Sarkar. 2009. Chunking Parallel Loops in the Presence of Synchronization *(ICS '09)*. Association for Computing Machinery, 181–192. https://doi.org/10.1145/1542275.1542304

[33] Rahul Shrivastava and V. Krishna Nandivada. 2017. Energy-Efficient Compilation of Irregular Task-Parallel Loops. *ACM TACO* 14, 4, Article 35 (nov 2017), 29 pages. https://doi.org/10.1145/3136063

[34] S. Subramaniam and D. L. Eager. 1994. Affinity Scheduling of Unbalanced Workloads. In *SC*. IEEE Press, 214–226.

[35] Peter Thoman, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer. 2012. Automatic OpenMP Loop Scheduling: A Combined Compiler and Runtime Approach. In *OpenMP in a Heterogeneous World*. 88–101.

[36] Ashkan Tousimojarad and Wim Vanderbauwhede. 2014. Comparison of Three Popular Parallel Programming Models on the Intel Xeon Phi. In *Euro-Par 2014: Parallel Processing Workshops*. Springer International Publishing.

[37] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed Power-law Graph Computing: Theoretical and Empirical Analysis. In *Advances in Neural Information Processing Systems*, Vol. 27. Curran Associates, Inc.

# A Appendix

```
1  Function getWL(Node X):
       // Returns an expression (as a string) constaining an estimate of the work-load of one
          iteration of OpenMP parallel-for-loop
2      int staticC = 0; // input independent
3      String dynamicC = "0"; // input dependent
4      foreach S ∈ X.statements:
5          case S is a loop:
6              String lboundC = getLoopBound(S);
7              String predC = predEvalCost;
8              String lbodyC = getWL(S.body);
9              dynamicC = dynamicC || "+" || lboundC || "*(" || predC || "+" || lbodyC || ")";
10         case S is an if-then-else statement:
11             String predC = predEvalCost;
12             String thenC = getWL(S.thenBody);
13             String elseC = getWL(S.elseBody);
14             dynamicC = dynamicC || "+" || predC || "+" || MAX(thenC, elseC);
15         case Otherwise:
16             staticC = staticC + S.staticC
17     return dynamicC || "+" || string(staticC);  // CostExpr
```

Fig. 24. Function to calculate the workload of each iteration of OpenMP parallel-for-loop.

For the sake of completeness, we restate the important aspects of the cost-expression generator scheme of Shrivastava and Nandivada [33, Section 3.1, Fig. 7] to generate the cost expression (`CostExpr`) for the body of each parallel-for-loop; interested readers may see the complete details in their paper.

The function `getWL` maintains the cost-expression of the parallel for loop in two parts: an input dependent part (in a string variable dynamicC), and an input-independent part (in an integer variable staticC). The final expression denoting the cost of the iteration of the parallel-for-loop is obtained by concatenating dynamicC and staticC; we use the || operator to concatenate two strings. The function iterates over all the statements of the parallel for loop. If the statement is a loop, `getWL` generates a string denoting an expression that multiplies the loop-bound, to the cost of the loop iteration (including the loop-predicate cost and loop body), and appends this string to dynamicC. This multiplication ensures that two iterations executing the same syntactic piece of code with different values for the loop-bound expressions will have different workloads at runtime. The function getLoopBound tries to identify (and return as a string) a closed-form expression denoting the loop bound. If the loop-bound expression cannot be computed (for example, for a while-loop), then getLoopBound conservatively returns a special constant CL. If any of the called functions (getLoopBound, or getWL) returns CL, then getWL returns CL. This special casing is not explicitly shown in Fig. 24, for ease of explanation. If the statement is a conditional, `getWL` generates an expression to compute the conservative worst-case cost of the statement (including the cost of evaluating the predicate). The macro $MAX(Y1, Y2)$, expands to "("||$Y1$||" ≥ "||$Y2$||")?"||$Y1$||" : "$Y2$. If the current statement is a simple statement (neither a loop nor a conditional), then `getWL` increments staticC by the static cost of the current statement.