COWS for High Performance!

Cost Aware Work Stealing for High Performance

***Abstract*—**

Parallel languages 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 manuscript, 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 (work-stealing based on the number of remaining iterations) and WSRW (work-stealing 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. Over five different benchmark programs, using five different input datasets, on two different hardware across a varying number of threads, we show that our approach achieves clear performance gains compared to the default scheduling schemes supported by OpenMP.

I. Introduction

With multicore systems going mainstream, efficient parallel computing has become a necessity than a choice. One pervasive pattern seen in modern parallel applications is loop-level parallelism. Almost all the modern languages, like OpenMP [1] Cilk [2], Chapel [3], HJ [4], X10 [5], and others have builtin 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 iterations of the loop are distributed among the runtime threads (also known as workers); this problem is commonly known as the loop-scheduling problem. One of the common targets of loop-scheduling is to ensure that load-imbalance between the threads is minimized.

In regular loops, where work done in each iteration is more or less independent of the input (and hence each iteration does nearly the same amount of work), achieving a good load balancing is relatively easy. However, when with irregular loops, the loop scheduling problem becomes very challenging.

Figure 1 shows a typical pattern (in OpenMP) found in many IMSuite [6] kernels and many graph analytics kernels. The `omp parallel` pragma leads to the creation of a team of threads, to concurrently execute the parallel-region (the body of `omp parallel`). The `omp for` pragma defines a work-sharing construct (parallel-for-loop) that is used to distribute the iterations of the parallel-for-loop among the team of threads (using a user chosen scheduling method). For example, 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 VI) the degree of vertices varies between 1 to 28800. This is a typical case with all the power-law graphs.

```
 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.

There have been various approaches proposed in the past to tackle the problem loop-scheduling for irregular parallel loops. Some of the impactful approaches include, static scheduling [1], loop-chunking [7], deep chunking [8], dynamic and guided scheduling [1], work-stealing [2], 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. While the dynamic and guided scheduling schemes of OpenMP and the popular work-stealing scheme allows 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. [7] 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.

To address these issues, in this manuscript, 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.

Contributions:

• We propose a new technique called COWS for load balancing irregular parallel loops. It uses a mixed compile-time and runtime approach to estimate the remaining workload and steal from an appropriate victim based on the remaining workload.

• 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 work (list of iterations) for each thread.

• We have implemented the static compiler component of COWS in the IMOP compiler framework and the runtime component in libgomp a shared library for OpenMP in GCC. We have studied the performance of COWS on five 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 studied the impact of COWS in the context of roughly-regular graphs and find that COWS gives reasonable performance improvement over the rest.

II. Background

Scheduling schemes in OpenMP. OpenMP supports four scheduling policies (static, dynamic, guided, and runtime) to map a loop iteration `i` to one of the `T` threads. (i) `static` schedule: scheduling method specified as `static,[c]`. The programmer may pass an optional argument `c >= 1`, that specifies the chunk size and the iteration `i` gets mapped to which thread: `(i / c)%T`. We term a special case of static scheduling when `c = 1`, as `cyclic` chunking. When the option `c` is not passed iteration `i` is mapped to thread `i/( N/ T)`; we use `static` schedule to denote this default scheduling policy. (ii) `dynamic` schedule: scheduling method is specified as `dynamic,[c]`. Every time a thread is idle, it is assigned at most `c` (default value = 1) iterations. (iii) `guided` schedule: scheduling method is specified as `guided,[c]`. Every time a thread is idle, it is assigned max(`c`, 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 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`.

III. Cost Aware Work Stealing

The traditional work-stealing schedulers, despite their advantages, suffer from multiple issues: (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 work on a list of explicit 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.

A. Efficiently Maintaining the Assigned Work

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 any thread t_i can be represented as a three tuple $\langle t_j, x_i, y_i \rangle$, where x_i indicates the index of the iteration in the initial-iteration-list of t_j, and y_i indicates the number of remaining iterations to be executed from the initial-iteration-list of t_j starting from x_i.

B. Control flow of a thread

Figure 2 depicts the flowchart of the overall control flow of any thread. For efficiency, we allow a parameter c 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 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, boxes with the bold boundaries indicate the critical sections.

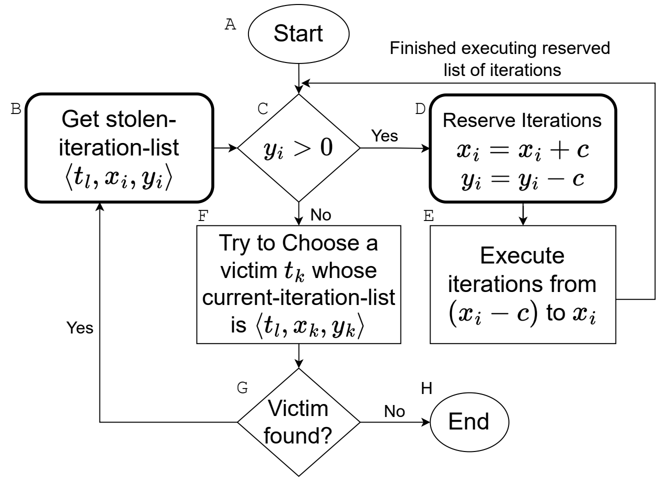

A
Start
Finished executing reserved list of iterations
B
Get stolen-iteration-list $\langle t_l, x_i, y_i \rangle$
C
$y_i > 0$
Yes
D
Reserve Iterations
$x_i = x_i + c$
$y_i = y_i - c$
No
F
Try to Choose a victim t_k whose current-iteration-list is $\langle t_l, x_k, y_k \rangle$
E
Execute iterations from $(x_i - c)$ to x_i
Yes
G
Victim found?
No
H
End

Fig. 2: 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.

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). 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 III-E. Each thread continues this cycle, till it finds a victim thread to steal iterations from.

Example. Say, there are 10 iterations in a parallel-for-loop, there are 3 threads at runtime, and c = 1. Figure 3 shows how different threads maintain their iteration lists. 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_3 completes all the remaining iterations while other threads are still executing their first iteration; Row 5 shows the current-iteration-lists. Now t_3 becomes a thief and will try to choose a victim. Say it chooses t_1 as the victim and successfully steals work (half the iterations = 1 iteration) from it. Row 6 shows the current-iteration-lists after the stealing. Say, 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. While a scheme that chooses the victim randomly has to bear the cost of random number generation only in terms of overheads for victim selection, the efficiency of such a scheme takes a big hit due to the possibly large overheads arising out of a large number of times the thieves steal the iterations. We call such a scheme WS-Random (WSR, in short).

<table>
<tr><td>1.</td><td></td><td>t_0</td><td>t_1</td><td>t_2</td></tr>
<tr><td colspan="5"><em>After initial cyclic distribution.</em></td></tr>
<tr><td>2.</td><td>List of iterations</td><td>[0, 3, 6, 9]</td><td>[1, 4, 7]</td><td>[2, 5, 8]</td></tr>
<tr><td>3.</td><td>initial-iteration-list</td><td>$\langle 0, 0, 4 \rangle$</td><td>$\langle 1, 0, 3 \rangle$</td><td>$\langle 2, 0, 3 \rangle$</td></tr>
<tr><td colspan="5"><em>After each thread has reserved c iterations, for itself.</em></td></tr>
<tr><td>4.</td><td>current-iteration-list</td><td>$\langle 0, 1, 3 \rangle$</td><td>$\langle 1, 1, 2 \rangle$</td><td>$\langle 2, 1, 2 \rangle$</td></tr>
<tr><td colspan="5"><em>t_3 finished all the initially assigned iterations, other threads still executing iteration#1.</em></td></tr>
<tr><td>5.</td><td>current-iteration-list</td><td>$\langle 0, 1, 3 \rangle$</td><td>$\langle 1, 1, 2 \rangle$</td><td>$\langle 2, 3, 0 \rangle$</td></tr>
<tr><td colspan="5"><em>After t_3 has stolen one iteration from t_1.</em></td></tr>
<tr><td>6.</td><td>current-iteration-list</td><td>$\langle 0, 1, 2 \rangle$</td><td>$\langle 1, 1, 2 \rangle$</td><td>$\langle 0, 3, 1 \rangle$</td></tr>
<tr><td colspan="5"><em>All threads finished assigned iterations; no more stealing.</em></td></tr>
<tr><td>7.</td><td>current-iteration-list</td><td>$\langle 0, 3, 0 \rangle$</td><td>$\langle 1, 3, 0 \rangle$</td><td>$\langle 0, 4, 0 \rangle$</td></tr>
</table>

Fig. 3: Illustration of how the threads maintain their iteration lists.

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.

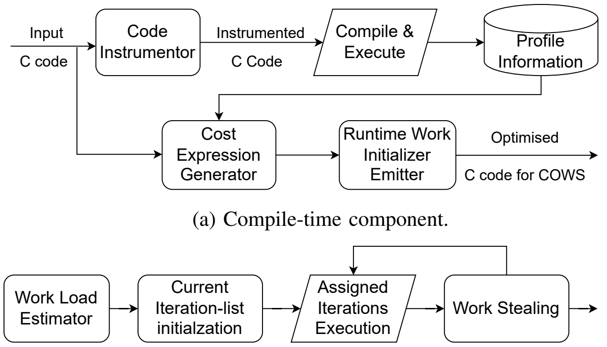
C. Remaining Iterations as the Estimate for Remaining Work

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 to obtain the victim with the maximum number remaining-iterations. We call such a scheme WS-Remaining Iterations (WSRI, in short).

D. 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 `i`th node. Thus, the cost of each iteration `i` depends on the degree of the `i`th node and cannot be estimated statically. To address such issues, inspired by prior works that recognize such a difficulty [8], [9], 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.



(a) Compile-time component.

(b) Runtime-time component.

Fig. 4: Overall technique

1) Compile-time Component.

Figure 4 shows the block diagram of our overall process. 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 [9]. The next phase of compilation (shown in Figure 4a) 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 [9, Section 3.1, Figure 7] to generate the cost expressions for the body of each parallel-for-loop; interested reader may see the details given by them. 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 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 [9] 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 we fall back to WSRI for that parallel loop.

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 5 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 an array `TArr[tid]` of long-integers, such that `TArr[tid][i]` will hold the cost of the `i`th iteration allotted to thread `tid` (like shown in the commented line 8). 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 `TArr[`t`][`$x+y$`]`$-$`TArr[`t`][`x`]`.

```
 1 int tid = omp_get_thread_num();
 2 long chunk = loopBound/T + 1;
 3 TArr[tid]=realloc(TArr[tid],chunk*sizeof(long));
 4 int counter = 0;
 5 long result = 0;
 6 #pragma omp for schedule(static,1)
 7 for (int i = 0; i < loopBound; i++) {
 8         // ~~result~~ = ~~CostString~~;
 9        result += CostString;
10        TArr[tid][counter++] = result;
11 }
```

Fig. 5: Runtime workload-estimator code, emitted just before a parallel-for-loop, whose for-loop is of the form `for(i=0;i<loopBound;i++){...}`.

Example. For the code shown in Figure 1, the emitted workload-estimator code can be obtained by replacing the variable `loopBound` with `g->N` in Figure 5.

2) Runtime Component.

Figure 4b 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 5).

Each OpenMP parallel-for-loop gets translated (by GCC) into three parts : (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 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 algorithm.

E. Cost Aware Work Stealing Algorithm

We now discuss our work-stealing algorithm (shown in Figure 6), which implements the work-stealing part of the flow-chart shown in Figure 2. Our algorithm has four interesting properties: (i) the algorithm may be invoked by multiple threads at the same time; (ii) while thieves are trying to steal, the victim is locked for a minimal amount of time; (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

```
1  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                                    ⎪ Critical section
5          Unlock ( $global$ ) ;                            ⎬ for all thieves.
6          return FAIL;                                     ⎪
7      busyStatus[$v$] = true ;                             ⎪
8      Unlock($global$) ;                                   ⎭
9      Lock ( $local_v$ ) ;                                 ⎫
10     if (stealPt=WorkDivider($v$)) == 0 then              ⎪
11         Unlock( $local_v$ ) ;                            ⎪ Critical section for
12         return FAIL;                                     ⎬ thief and victim.
13     Say the current-iteration-list of $v$ is $\langle t_j, x, y\rangle$;    ⎪
14     Set the current-iteration-list of $v$=$\langle t_j, x, \text{stealPt}\rangle$);  ⎪
15     Unlock($local_v$) ;                                  ⎭
16     busyStatus[$v$] = false ;
17     Set the current-iteration-list of thief =
       $\langle t_j, \text{stealPt+1}, y - \text{stealPt}\rangle$ ;
18     return SUCC;
```

Fig. 6: Work-Stealing algorithm

```
1  Function findVictim (ThreadId $id$) : //Returns victim-id or
   -1
2      $victim$ = -1; $work_{max}$ = 0;
3      for $i = 0$ to T-1 do:
4          if busyStatus[$i$] $== true$ then continue ;
5          Say the current-iteration-list of $t_i$ = $\langle t_j, x, y\rangle$;
6          $work$ = TArr[$j$][$x + y$] - TArr[$j$][$x$] ;
7          if $work > work_{max}$ then
8              $work_{max} = work$;
9              $victim = i$;
10     return $victim$;
```

Fig. 7: WSRW victim finding scheme. T = # threads.

example, for WSRI, it invokes a code as described in III-C. For WSRW, Figure 7 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 6).

In Figure 6, 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 (Figure 2, box D). We distribute the iterations as per the Work Divider function.

WorkDivider *algorithm.* The details of this algorithm varies based on the strategy of work-stealing. The Algorithm 8 give 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 (stealPoint), such that the iterations of the victim, from x to stealPoint, have at least Threshold (set to half the remaining workload) amount of workload. The remaining iterations (from stealPoint+1 to y) can be stolen; the algorithm returns stealPoint. In case of WSRI, the WorkDivider simply returns $y - y/2$ (uses integer divison), if the current-iteration-list of the victim is $\langle t_j, x, y\rangle$.

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

Fig. 8: Work divider for the WSRW scheme.

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

Fig. 9: Illustrative pseudo code for a topology driven algorithm.

In Figure 6, 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.

IV. Optimizations

Opt1: Redundant Initialization. The WSRW scheme proposed in Section III, emits code to populates the TArr before every parallel-for loop. Many times it happens that the same parallel-for loop is run consecutively for more than one time; for example, till a fixed point is reached. 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 reuse the previously populated values.

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.

Opt3: Loop bodies guarded with a predicate. A common 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 9 shows an illustrative pseudo code showing a similar behavior. Our proposed cost estimation scheme (Section III-D) computes the cost of an if-else statement as the maximum cost of the then-block and else-block and consequently, may lead to highly in accurate 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 6, goes through all the remaining iterations of a thread to identify the value of the stealPoint. 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 stealPoint. Instead of sequentially iterating from x to $x + y$ (in Figure 8), 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 can be calculated using $\texttt{TArr}[(2 * x + y)/2 + 1] - \texttt{TArr}[x]$ and $\texttt{TArr}[y] - \texttt{TArr}[(2 * x + y)/2]$, respectively. If they are exactly dividing the workload into two halves then we return middle point as stealPoint. Otherwise, we repeat this divide and conquer strategy to find the partition such that the iterations of the victim, from x to stealPoint, 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 8), where N is the number of iterations of the parallel-loop.

V. Discussion

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 II)) 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. The reason why static and the cyclic schedule work so much better is because OpenMP provides a very highly tuned implementation of these schemes that involve pretty much zero overhead (does not involve locks, or invoke the scheduler, and so on). Our evaluation also establishes the supremacy of the static and cyclic schemes over the other OpenMP schedules.

VI. Implementation and Evaluation

As discussed in Section III, our proposed techniques have two components: the compile-time component and the runtime component. We have implemented our proposed compile-time component in IMOP [10], an open-source compiler framework (source-to-source) for OpenMP programs that makes it convenient to write compiler passes. Our implementation spanned $1.25k$ lines of Java code. We have implemented our proposed runtime component in libgomp, a shared library for OpenMP in GCC; this runtime component spanned $1.4k$ 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 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 inefficient.

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. 10: Characteristics of the datasets used.

Bench	LOC	SL	DA	DB	DC	DD	DE
1. KC	403	6	28	28	28	28	28
2. LE	269	3	34	30	16	1288	1044
3. BFS	231	2	15	15	9	556	543
4. PR	184	2	30	26	52	44	44
5. CC	170	3	7	9	4	243	249

Fig. 11: Benchmarks used for evaluations Abbreviations: LOC: Lines of code; SL: #static 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.

Our proposed techniques were evaluated on five popular shared-memory kernels shown in Figure 11: K-committee (KC); leader election for general graphs (LE); breadth first search (BFS); page rank (PR); and connected components (CC). The source code for the first three kernels are taken from IMSuite [6] and remaining two were written based on kernels used in prior work [11]. Figure 11 also lists some characteristics of these kernels. These include, number of lines of code, number of static parallel-for-loops, and the number of dynamic parallel-for-loops encountered in the context of the five above discussed inputs. 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) [12]: Youtube, LiveJournal and Orkut graphs. Besides testing the efficacy of our proposed system in the presence 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 10 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 (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. Due to the lack of space, we skip showing the graphs for the evaluation on Goodwill and only present the summary.

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 to be more or less the best among the default schedules supported by OpenMP. 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 four dimensions: (i) Performance gains in the context of highly irregular graphs. (ii) Behavior in the context of roughly regular graphs. (iii) Impact of proposed optimizations. (iv) Reduction in the number of steals compared to the traditional techniques.

A. 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 under consideration.

KC. Figure 12 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).

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 III-D1).

As it is well known, among the default schedules provided by OpenMP, the cyclic schedule performs quite well, and it even outperformed the WSR slightly. 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). We have observed more or less similar hardware based behaviours on the other four kernels as well.

LE. Figure 13 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. On LE, we can more or less find a clear order among the different scheduling schemes: WSRW, WSRI, cyclic, WSR, guided, static, dynamic.

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

BFS. Figure 14 shows the speedups obtained for the BFS kernel. Overall we see that for the BFS kernel, WSRW, WSRI and cyclic more or less outperform the rest. 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 more or less matched due the application of Opt3 (Section IV) on the main loop.

Compared to the cyclic schedule, for varying number of cores and across all the three inputs, WSRW (and WSRI) was 1.09× and 0.95× better on Goodwill and Aqua 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, while WSRI and WSRW pay the respective overheads, the remaining work estimate is still not accurate. Thus we do not get much gains compared to cyclic (a highly efficient scheme of scheduling with minimal overheads; see Section V). However, WSRW and WSRI perform better than the remaining scheduling schemes.

PR. Figure 15 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. Similar to LE, we can more or less find a clear order among the different scheduling schemes.

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

CC. Figure 16 shows the speedups obtained for the CC kernel. Overall we see that for the CC kernel, even though WSRW more or less 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

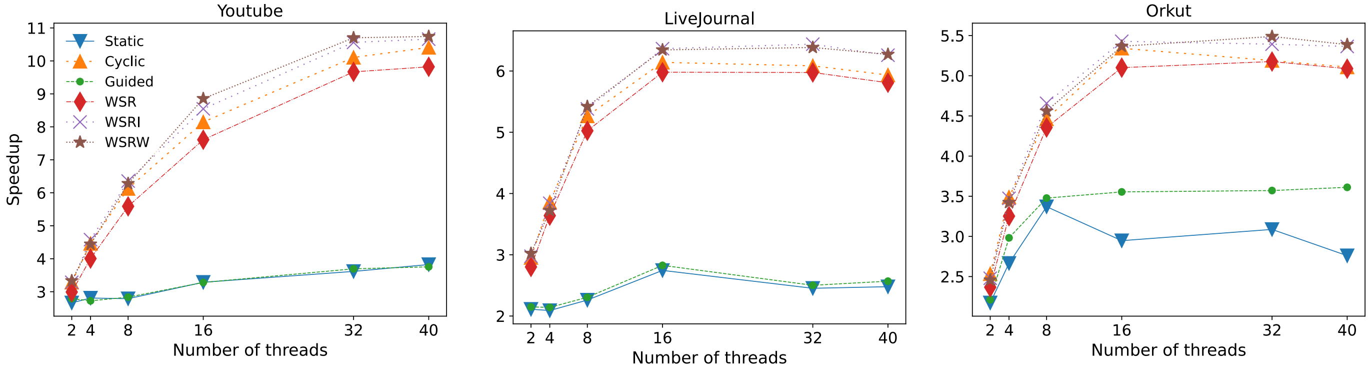


Fig. 12: Kernel = KC. Speedups with respect to dynamic scheduling. The points have been connected only to improve the visibility.

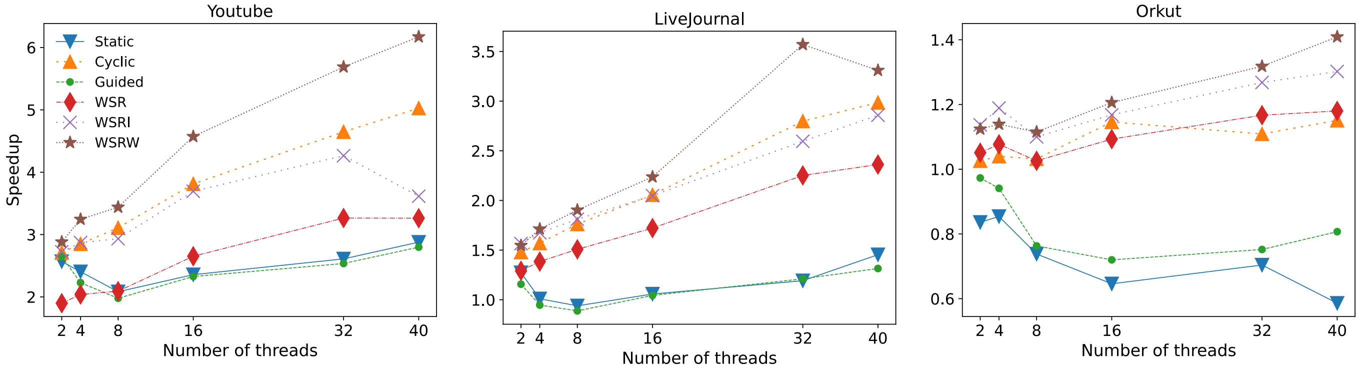


Fig. 13: Kernel = LE. Speedups with respect to dynamic scheduling. The points have been connected only to improve the visibility.

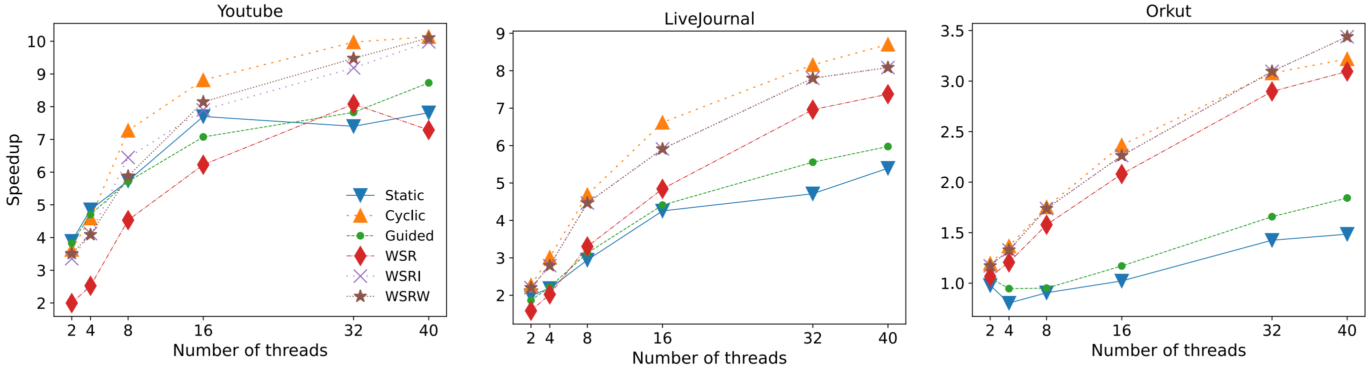


Fig. 14: Kernel = BFS. Speedups with respect to dynamic scheduling. The points have been connected only to improve the visibility.

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 1.12× and 1.06× better on Goodwill and Aqua system, respectively. Similarly, compared to the cyclic-schedule, WSRI was 1.08× and 1.03× better on Goodwill and Aqua 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 on average (geomean) WSRW and WSRI get 1.10× and 1.05× performance over the cyclic schedule (the best among the default schedules provided by OpenMP).

B. 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 impact (possibly negative?) 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 10). For brevity, we only show the performance results for the maximum number of threads available on the

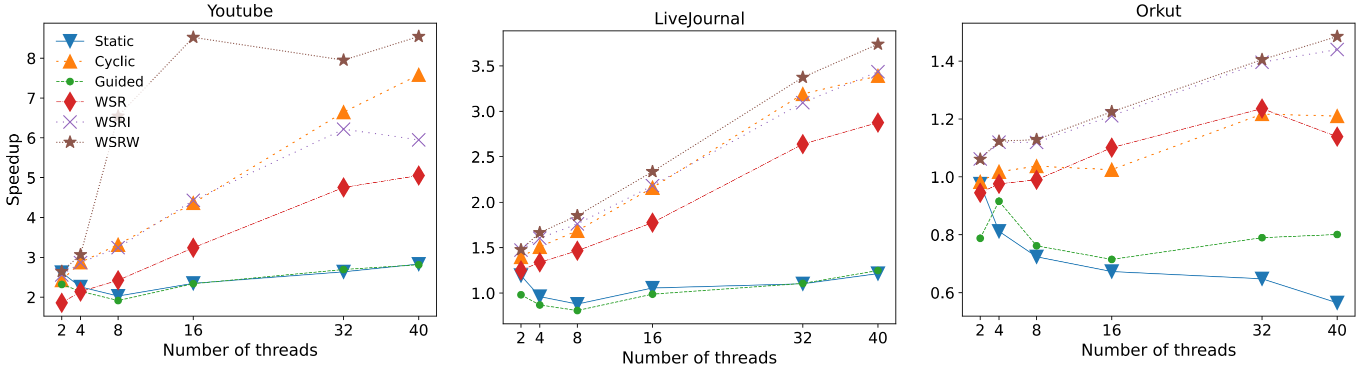


Fig. 15: Kernel = PR. Speedups with respect to dynamic scheduling. The points have been connected only to improve the visibility.

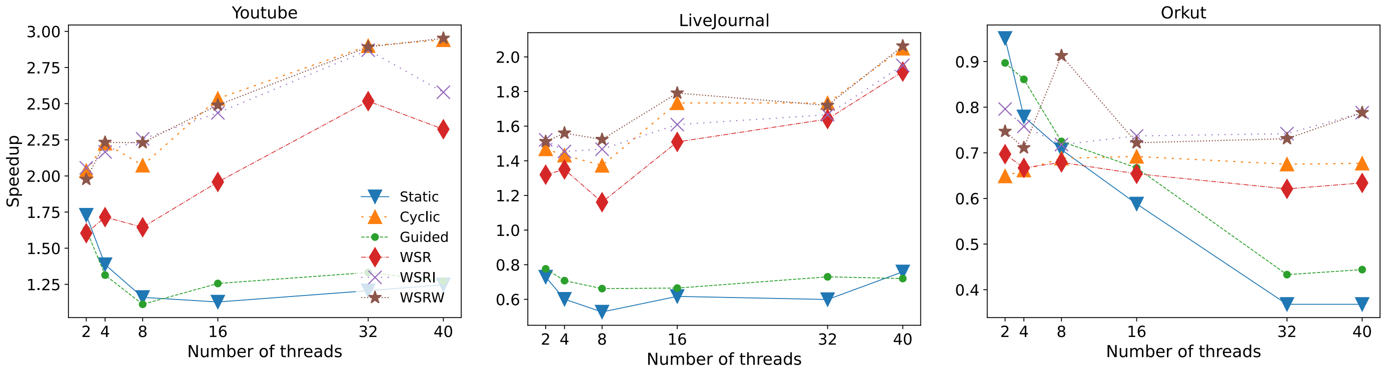
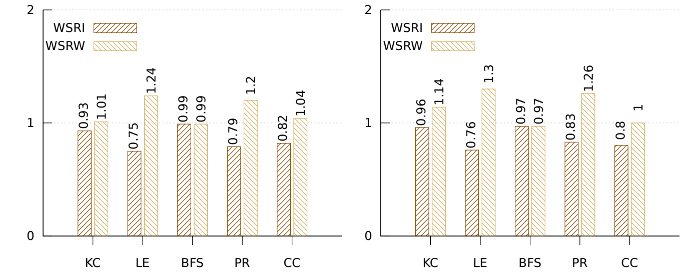


Fig. 16: Kernel = CC. Speedups with respect to dynamic scheduling. The points have been connected only to improve the visibility.



(a) RoadNet-CA

(b) RoadNet-PA

Fig. 17: Speedup of WSRW and WSRI over cyclic-scheduling on roughly regular graphs. Number of threads = maximum # cores.

respective hardware systems.

The Figure 17 shows the performance of WSRW and WSRI shown as speedups over cyclic-scheduling - the best among the remaining scheduling techniques, discussed above; 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. Across the two datasets and both the hardware, we find that WSRW obtains speedups of 1.07×, 1.34×, 0.96×, 1.17×, and 1×, for KC, LE, BFS, 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×, 0.92×, and 0.90×, for KC, LE, BFS, 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.

C. Impact of Optimizations

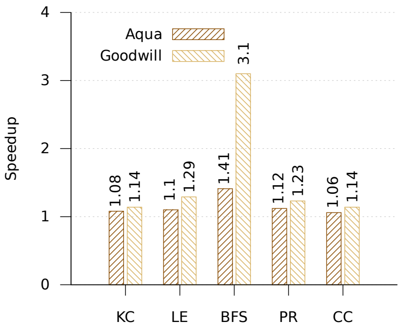


Fig. 18: Speedup of optimized code Vs unoptimized. Number of threads = max number of cores on the respective system.

To study the impact of the proposed optimizations (Section IV) we have compared the performance of WSRW, with and without the proposed optimizations. Figure 18 shows the geometric mean performance gains (across all the five datasets) due to our proposed optimizations, when the kernels are run using the maximum number of available hardware cores on each system. We see that the impact due to the optimizations is significant: 1.14× to 3.1× on Goodwill and 1.06× to 1.41× on Aqua.

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. We also show that the proposed optimizations are impactful.

VII. Related Work

The OpenMP API supports static, dynamic, and guided scheduling. OpenMP chunking policies are unconcerned about the input program structure or parameters. These existing schemes of OpenMP are insufficient for improving the performance of various applications and systems [13].

Loop-scheduling. There have been many works that go beyond the default set of scheduling schemes (static, dynamic, guided) supported by OpenMP [1], [14]–[16]. Especially considering the challenges in scheduling irregular parallel loops, there have been many focused attempts in this direction. BinLPT [17] 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. [18] 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 [8] 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.

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. [19] use profiling to improve the mapping of processes to different processors and Shrivastava and Nandivada [9] use profiling to migrate the threads and scale the frequency of processors for saving energy. Similarly, Kejariwal et al. [20] and Bull [21] 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 [2], [22]–[24]. Recently [25] present a work-stealing 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 static/cyclic – 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, that has very low overheads (no queues) and performs better than all the schedules supported by OpenMP (including static and cyclic).

VIII. 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 five different kernels, using five different input datasets, on two different hardware, for varying number of threads and show that COWS gives the best of both worlds: the low overheads of static scheduling and dynamic load balancing of work-stealing.

References

[1] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.0," May 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *SIGPLAN Not.*, vol. 30, no. 8, p. 207–216, aug 1995.

[3] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *IJHPCA*, vol. 21, no. 3, p. 291–312, 2007.

[4] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: The New Adventures of Old X10." ACM, 2011, p. 51–61.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing," *SIGPLAN Not.*, vol. 40, no. 10, p. 519–538, oct 2005.

[6] S. Gupta and V. K. Nandivada, "IMSuite: A benchmark suite for simulating distributed algorithms," *JPDC*, vol. 75, no. 0, pp. 1–19, 2015.

[7] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar, "A Transformation Framework for Optimizing Task-Parallel Programs," *TOPLAS*, vol. 35, no. 1, apr 2013.

[8] I. K. Prabhu and V. K. Nandivada, "Chunking Loops with Non-Uniform Workloads," in *ICS*. ACM, 2020.

[9] R. Shrivastava and V. K. Nandivada, "Energy-Efficient Compilation of Irregular Task-Parallel Loops," *ACM TACO*, vol. 14, no. 4, nov 2017.

[10] A. Nougrahiya and V. K. Nandivada, "IIT Madras OpenMP (IMOP) Framework," 2018.

[11] A. Rajendran and V. K. Nandivada, "DisGCo: A Compiler for Distributed Graph Analytics," *TACO*, vol. 17, no. 4, sep 2020.

[12] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, Jun. 2014.

[13] F. M. Ciorba, C. Iwainsky, and P. Buder, "OpenMP Loop Scheduling Revisited: Making a Case for More Schedules," in *Evolving OpenMP for Evolving Architectures@IWOMP*, 2018.

[14] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Practical and Robust Method for Scheduling Parallel Loops," in *SC'91*, p. 610–632.

[15] C. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE TOSEM*, vol. SE-11, no. 10, pp. 1001–1016, 1985.

[16] S. Lucco, "A Dynamic Scheduling Method for Irregular Parallel Programs," *SIGPLAN Not.*, vol. 27, no. 7, p. 200–211, jul 1992.

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

[18] P. Thoman, H. Jordan, S. Pellegrini, and T. Fahringer, "Automatic OpenMP Loop Scheduling: A Combined Compiler and Runtime Approach," in *OpenMP in a Heterogeneous World*, 2012, p. 88–101.

[19] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, "MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters," in *ICS*, 2006.

[20] A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos, "History-aware Self-Scheduling," in *ICPP'06*, pp. 185–192.

[21] J. M. Bull, "Feedback Guided Dynamic Loop Scheduling: Algorithms and Experiments," in *EuroPar*, 1998, pp. 377–382.

[22] R. H. Halstead, "MULTILISP: A Language for Concurrent Symbolic Computation," *TOPLAS*, vol. 7, no. 4, p. 501–538, oct 1985.

[23] R. D. Blumofe and D. Papadopoulos, "Hood: A User-Level Threads Library for Multiprogrammed Multiprocessors," Tech. Rep., 1998.

[24] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," *SIGPLAN Not.*, vol. 33, no. 5, p. 212–223, may 1998.

[25] J. Dennis Booth and P. Allen Lane, "An adaptive self-scheduling loop scheduler," *CCPE*, vol. 34, no. 6, p. e6750, 2022.