2016 6th Workshop on Irregular Applications: Architecture and Algorithms

Optimized Distributed Work-Stealing

Vivek Kumar
Rice University

Karthik Murthy
Rice University

Vivek Sarkar
Rice University

Yili Zheng
Lawrence Berkeley National Laboratory

***Abstract*—Work-stealing is a popular approach for dynamic load balancing of task-parallel programs. However, as has been widely studied, the use of classical work-stealing algorithms on massively parallel and distributed supercomputers introduces several performance issues. One such issue is the overhead of *failed steals* (communicating with a victim that has no work), which is far more severe in the distributed context than within a single SMP node. Due to the cost of inter-node communication, it is critical to reduce the number of failed steals in a distributed context. Prior work has demonstrated that load-aware victim processor selection can reduce the number of failed steals, but it cannot eliminate the failed steals completely.**

In this paper, we present two different load-aware implementations of distributed work-stealing algorithm in HabaneroUPC++ PGAS library — BaselineWS and SuccessOnlyWS. BaselineWS follows prior work in implementing a distributed work-stealing strategy. SuccessOnlyWS implements a novel distributed work-stealing strategy that completely eliminate inter-node failed attempts by introducing a new policy for moving work from busy to idle processors. This strategy also avoids querying the same processor multiple times with failed steals. We evaluate both BaselineWS and SuccessOnlyWS by using up to 12288 cores of Edison, a CRAY-XC30 supercomputer and by using dynamic irregular applications, as exemplified by the UTS and NQueens benchmarks. We demonstrate that SuccessOnlyWS provides performance improvements up to 7% over BaselineWS.

***Index Terms*—Distributed work-stealing; Habanero, PGAS;**

I. Introduction and Background

Work-stealing [1] is a popular load balancing technique for dynamic task-parallelism. It maintains a pool of *workers*, each of which maintains a double-ended queue (*deque*) of tasks. When local deque becomes empty, worker becomes a *thief* and seeks a *victim* from which to *steal* work. However, it may happen that the victim has run out of work when it receives the steal attempt from the thief. This situation is called a *failed steal attempt*, as the thief did not receive any task.

Today almost all supercomputers are built from multi-core processors connected via high speed interconnects, offering a mixture of shared memory and distributed memory parallelism. Prior studies have demonstrated the benefits of using a hybrid-programming model based distributed work-stealing runtimes [2], [3]. These runtimes employ one process per node, each containing multiple worker threads. The benefits include improving memory utilization per node, computation-communication overlap, and scalability. One dedicated thread (communication worker) takes the role of managing all inter-node communications, while the rest of the threads (computation workers) participate in local computations and steal among themselves using low-overhead compare-and-swap operations. When computation workers run out of work within a process, they contact their communication worker to request initiation of distributed work-stealing by communicating with remote (victim) communication workers. The victim communication worker can either maintain a ready queue of tasks to send to the remote thief [3], or lazily performs intra-node steals to send tasks to the remote thief [2].

While intra-node failed steals only waste the thief's CPU cycles, inter-node work-stealing wastes the victim's CPU cycles as well (e.g., when the communication worker in the victim process performs lazy intra-node steals). It is critical to reduce the number of failed steals in a distributed context due to the cost of inter node communication. Techniques such as load-aware victim selection (by using network RDMA to estimate total tasks at victim [4]) and lifeline graphs [5] can reduce the number of failed attempts but cannot completely avoid the issue.

In this paper, we analyze inter-node failed steals as a source of performance degradation in a hybrid-programming model based distributed work-stealing. With inspiration from the lifeline graphs [5], we introduce a new policy to choose a remote victim that completely avoids the failed steals at inter-node level. To the best of our knowledge, this paper is the first to completely remove the inter-node failed steals in distributed work-stealing.

In summary, this paper makes the following contributions:

- Two different implementations of distributed work-stealing in HabaneroUPC++ PGAS library — BaselineWS and SuccessOnlyWS.
- We implement BaselineWS using traditional approaches to distributed work-stealing algorithm, whereas in SuccessOnlyWS we introduce a new victim selection policy that completely avoid all inter-node failed steal attempts.
- We demonstrate the benefits of SuccessOnlyWS over BaselineWS by scaling irregular computations up to 12288 cores on the Edison supercomputer at NERSC. Our evaluation shows that by avoiding inter-node failed steals attempts, SuccessOnlyWS deliver performance benefits up to 7% without ever degrading the performance.

II. Design and implementation

A. *Programming model for distributed work-stealing*

In our prior work, we introduced HabaneroUPC++ [6] as a compiler-free PGAS library that supports a tighter integration of intra-place and inter-place parallelism than standard hybrid programming approaches. It uses C++11 lambda-based user interfaces for launching asynchronous tasks, such as: a) **`async`**, **`asyncAwait`**, **`asyncPhased`** and **`forasync`** (at the intra-place level); b) **`asyncAt`** for asynchronous remote function invocation; c) **`asyncCopy`** for asynchronous local/remote copy; and d) a barrier style **finish_spmd** for joining all these asynchronous tasks. HabaneroUPC++ uses a dedicated communication worker per place.

Now, for supporting distributed work-stealing in HabaneroUPC++, we introduce a new user interface to launch locality flexible asynchronous tasks [3]. We call this new interface as **asyncAny** and it accepts a user defined C++11 lambda function representing a locality flexible task. **asyncAny** tasks can be arbitrarily nested and can be joined by using **finish_spmd**, similar to other tasks in HabaneroUPC++.

B. Load-aware victim selection in HabaneroUPC++

As in prior work [4], both BaselineWS and SuccessOnlyWS use load-aware inter-place victim selection. In both these runtimes, **asyncAny** tasks are free to execute anywhere in the cluster. Total **asyncAny** tasks at a place are the sum of **asyncAny** tasks at each of its computation workers. Communication worker at each HabaneroUPC++ place publishes current count of **asyncAny** tasks (at its place) in a shared variable in global address space (`placeLoad`). Remote places uses network RDMA to read this information to get an estimate of total number of **asyncAny** tasks at the destination place. Today most of the HPC platforms support RDMA. These RDMA operations do not require any remote CPU involvement and are very efficient.

Computation workers steal among themselves (intra-place steals) using low overhead compare-and-swap operations. Inter-place steals are routed only through the communication worker. A communication worker will become a thief if any of the computation workers at its place are idle.

C. Remote steal request from thief to victim

Algorithm 1 Remote steal request from thief to victim

```
 1: procedure STEAL_ASYNCANY
 2:     while $global\_termination\_detection \neq true$ do
 3:         $v \leftarrow$ GET_RANDOM_VICTIM_ID($total\_places$)
 4:         if $placeLoad[v] > THRESHOLD$ then
 5:             if $SuccessOnlyWS$ then
 6:                 if $already\_contacted[v] \neq true$ then
 7:                     ASYNCAT(v, queue_my_place)
 8:                     WAIT_UNTIL_ASYNCAT_INFLIGHT()
 9:             else                                    ▷ BaselineWS
10:                 if TRYLOCK($v$) then
11:                     $thief\_queued[v] \leftarrow my\_place$
12:                     WAIT_UNTIL_VICTIM_REPLY()
13:                     UNLOCK(v)
14:         if $asyncAny\_received$ then
15:             if $SuccessOnlyWS$ then
16:                 FORGET_VICTIM($asyncAny\_source$)
17:             $break$
```

Algorithm 1 shows the pseudocode for sending steal request from the thief to the victim for both BaselineWS and SuccessOnlyWS. Once the communication worker becomes a thief, it enters the global task search cycle (line 2) where it attempts to steal task from a potential victim (lines 3 and 4).

In BaselineWS, the thief will mark this remote victim as occupied (line 11) using a lock and a flag in victim's global address space. The thief can never choose a pre-occupied victim. Thief will wait until it receives the victim's response (line 12). The victim can either send **asyncAny** tasks to this thief or may deny the request if it has started experiencing shortage of **asyncAny** tasks (failed steal). The thief might even try same victim multiple times in failed steals. The thief will restart the entire procedure until it succeeds or it decides that all the victims are idle.

Unlike BaselineWS, in SuccessOnlyWS the thief neither uses a lock (line 10), nor marks a victim as occupied (line 11). A thief remembers the place id of all the victims it attempts to steal from (line 6). It will never steal from a victim, which is still registered in its memory. It will forget a victim only after it receives any task from it (line 16). Once a victim has been decided, the thief will send a steal request to this victim using **asyncAt** (line 7). This **asyncAt** terminates after executing at victim's place (communication worker) and pushing place id of the thief on the victim's memory. As this design does not use a lock (line 10), thieves will be registered at the victim in the order messages arrive at the victim. Also, multiple thieves can simultaneously send steal requests to a same victim. Similar to thief, victim also memorizes the place id of all the thieves who have requested tasks from it. Once the **asyncAt** has terminated (line 8), the thief will continue its task search cycle. It will hunt its next potential victim and repeats the entire procedure until: a) it either receives task (line 14) from any of the previously contacted victims (end of current task search cycle); or b) it is unable to find any potential victim.

In SuccessOnlyWS the thief wait until the **asyncAt** is in flight (line 8), whereas in BaselineWS the thief wait until it receives the victim's response (line 12). The cpu cycles spent by the thief while waiting will be wasted in BaselineWS for every failed steal attempt. However, this is not true for SuccessOnlyWS as every steal attempt is guaranteed to fetch **asyncAny** tasks. This wait is a necessary design choice in SuccessOnlyWS as otherwise thief may end up sending **asyncAt** (steal requests) to several remote victims.

D. Transferring tasks from victim to remote thief

The pseudocode for transferring tasks from victim to remote thief is shown in Algorithm 2, for both BaselineWS and SuccessOnlyWS. We don't take the approach of [3] to maintain a ready queue of **asyncAny** tasks at communication worker. This design may suffer when there is large number of computation workers as they will try to steal back from the communication worker (akin to work-sharing overheads). Hence, in HabaneroUPC++ the communication worker follows a lazy approach by stealing from local computation workers (line 4 and 13) only in case of a pending remote steal request. Maximum number of **asyncAny** tasks sent to remote thief is specified by a user defined environment variable.

Algorithm 2 Transferring tasks from victim to remote thief

```
1: procedure SEND_ASYNCANY
2:     if $SuccessOnlyWS$ then
3:         while $total\_queued\_thieves > 0$ do
4:             $count \leftarrow$ LOCAL_STEAL($task\_array$)
5:             if $count > 0$ then
6:                 $thief\_place \leftarrow$ POP_THIEF()
7:                 ASYNCAT(thief_place, task_array)
8:                 FORGET_THIEF(thief_place)
9:             else $break$
10:    else                                         ▷ BaselineWS
11:        $thief \leftarrow thief\_queued[my\_place]$
12:        if $thief \neq EMPTY$ then
13:            $count \leftarrow$ LOCAL_STEAL($task\_array$)
14:            ASYNCAT(thief, task_array, count)
15:            $thief\_queued[my\_place] \leftarrow EMPTY$
```

In BaselineWS, at any given time only one thief can steal from a particular victim (line 12). Remote communication worker (victim) can deny (failed steal) or send **`asyncAny`** tasks to the waiting thief (line 14). Unlike BaselineWS, in SuccessOnlyWS several thieves could be registered at victim's memory and none of these requests should fail. For each pending steal requests, communication worker (victim) in SuccessOnlyWS performs a round of local steal (line 4). If it were successful, it would pop a thief id from its memory (line 6) and send the tasks to the remote thief (line 7). It then forgets this thief (line 8) and repeat the entire procedure, unless either its running short of **`asyncAny`** tasks, or if there are no more pending steal requests. Remote thief also forgets this victim after receiving the tasks.

The only caveat in SuccessOnlyWS is the *steal cycle* formation, which could happen in following scenario. Thief sends multiple steal requests. However, by the time a steal request gets queued at a victim, the victim gets shortage of tasks at its place and himself becomes a thief. The original thief receives task from some other victim and now is sufficiently busy. The original victim (now a thief) queues its own steal request at this thief (now a victim) even though it is already having a pending steal request from this place. The maximum percentage of cyclic steals we observed was 0.2% (Figure 1(d)), which is insignificant. Also, note that cyclic steals is not a deadlock, since both these places will send/obtain tasks from others.

III. Experimental Evaluation

1) Methodology: We have used two different trees (T1WL and T3WL) of the UTS benchmark [7] and the NQueens (henceforth mentioned as "NQ") test for all our experimental evaluations. We remove all the work-stealing related code from the open-sourced UPC implementation of UTS [7] and use **`asyncAny`** to create HabaneroUPC++ UTS version. We found the best performing chunk sizes (-c) and polling intervals (-i) as 8 and 10 respectively for both trees. NQ's objective is to find a placement for N queens on an $N \times N$ chessboard such that none of the queens attacks any other. We used N=18. When the search tree depth becomes greater than 6, we stop creating new **`asyncAny`** tasks. We used the Edison supercomputer at NERSC for our experiments. It has two sockets per node and each socket has 12 cores. In HabaneroUPC++, we allocate one place per node where one core is dedicated to be a communication worker and the remaining 23 cores are computation workers.

2) Results: Figure 1(a) shows the execution time for all the three benchmarks using BaselineWS runtime. Recall that we used all the node resources. Hence, for each runtime, at 512 nodes the total number of cores used is 12,288.

Both BaselineWS and SuccessOnlyWS policies in HabaneroUPC++ take the same approach of load-aware remote victim selection, but they too hold chances to attempt remote steal from idle victims. This is a failed steal in BaselineWS but not in SuccessOnlyWS where the victim will eventually return tasks or terminate. Figure 1(b) shows the total number of inter-place failed steals in BaselineWS. We only report inter-place failed steal attempts as steals within a place is performed using negligible overhead compare-and-swap operations.

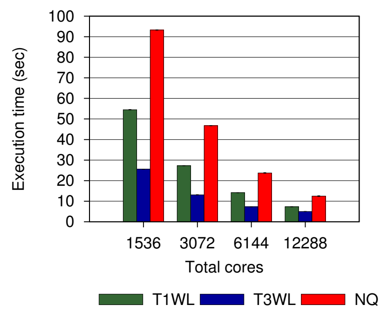
SuccessOnlyWS avoids any inter-place failed steal attempts and this has a direct impact on the performance improvements achieved by using SuccessOnlyWS. Figure 1(c) plots the speedup of SuccessOnlyWS over BaselineWS for all benchmarks. At 12288 cores by using SuccessOnlyWS, the benchmarks T1WL, T3WL and NQ executed 3%, 7% and 4% faster (respectively) than by using BaselineWS. It's worth noticing that SuccessOnlyWS never degrades the performance, even at smaller node counts.

In SuccessOnlyWS when a thief enters the global task search phase, it registers its place id on multiple victims until it receives task from any of the victims (end of a search phase). We performed analysis to understand the maximum number of victims that SuccessOnlyWS can attempt during its each task search phase. For this, we calculated the total number of victims contacted in each search phase of the thief. We then calculated the percentage of total search phases showing similar numbers of total victims contacted. Figure 1(d) shows the result of this experiment with SuccessOnlyWS using 12288 cores (similar results for other node values). We noticed that in more than 85% of cases, the thief contacts only two victims for all the benchmarks. Also, contacting more than four victims in any search phase is extremely rare for the thief.

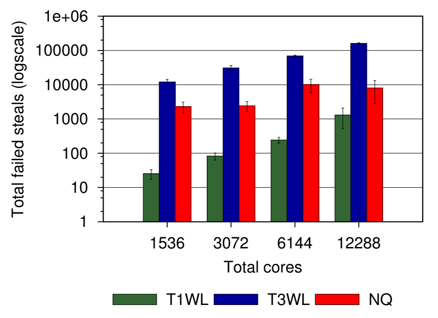
Section II-D explains the chances of steal cycle formations in SuccessOnlyWS. Figure 1(e) shows the total cyclic steals in SuccessOnlyWS as a percentage of total steals. The maximum percentage we observed was 0.2%, which is almost insignificant. We also implemented a variant of SuccessOnlyWS that would never allow any cyclic steals. In Figure 1(f) we compare the performance of SuccessOnlyWS at 12288 cores both with and without cyclic steals. We found both these versions of SuccessOnlyWS to perform similar.

IV. Related Work

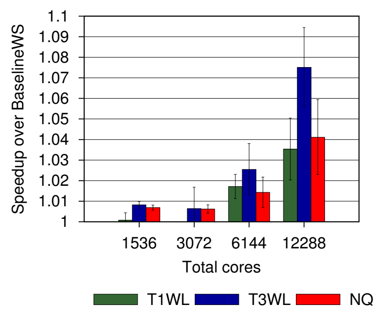
SuccessOnlyWS was inspired from lifeline based Global Load Balancing (GLB) [5] and hence shares some resemblance



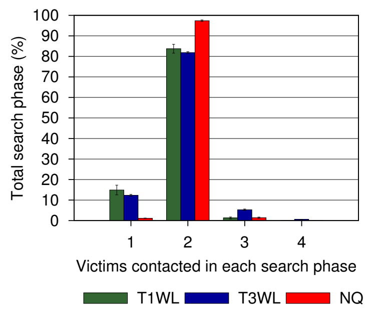
(a) Execution time using BaselineWS



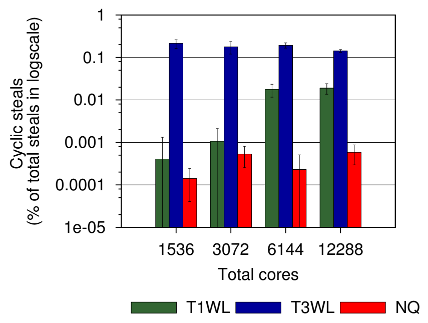
(b) Total inter-place failed steals in BaselineWS



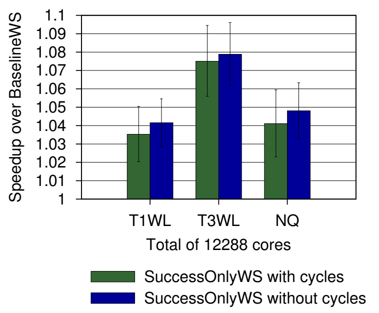
(c) SuccessOnlyWS speedup over BaselineWS



(d) Total remote victims contacted by thief in SuccessOnlyWS during each search phase (using 12288 cores)



(e) Steal cycle formations in SuccessOnlyWS (smaller is better)



(f) Speedup of SuccessOnlyWS over BaselineWS for both with and without cyclic steals (using 12288 cores)

Fig. 1. Experimental evaluation of BaselineWS and SuccessOnlyWS (strong scaling in each case)

to GLB. In GLB, once the program is launched, the runtime thief performs *two rounds* of searches. In the first round, it performs *w* random victim selections (*w* is subset of total number of place). If all *w* attempts results in failed steals, the thief attempts the second round in which it tries to steal from its lifeline buddies (victims). The total number (*z*) of lifeline buddies for each place is pre-computed using lifeline graphs (each place knows all its lifeline buddies). If the lifeline buddies are also idle, they remember incoming thief requests and send some work once they receive it.

SuccessOnlyWS differs from GLB as following: a) **`asyncAny`** based programming model in SuccessOnlyWS is simple and provides serial elision to sequential code, whereas in GLB the user application is extensively modified; b) unlike GLB, prior to attempting a steal from a remote victim, SuccessOnlyWS uses network RDMA based load-aware victim selection to maximize the success probability; c) thief in SuccessOnlyWS is free to attempt steal from any number of remote victims and its very rare for it to contact three or more victims (Section III) in any search phase (adaptively choosing lifelines, which are generally very few); d) GLB does not uses communication and computation workers based hybrid programming model as in SuccessOnlyWS.

V. Conclusion

We believe that work-stealing will be an increasingly important approach for effectively exploiting the performance of modern supercomputers, which tend to have more and more cores per node. However, work-stealing on a large scale cluster also introduces several performance issues due to the cost of inter-node communications. In this work we analyzed inter-node failed steal attempts as a source of performance degradation. We introduced a new design of distributed work-stealing, which completely removes the inter-node failed steal attempts and improves the performance by up to 7%.

Acknowledgment

This work is supported by the X-Stack program funded by the U.S. Department of Energy Office of Advanced Scientific Computing Research. Any opinions, findings and conclusions expressed herein are the author's and do not necessarily reflect those of the sponsors.

References

[1] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, Sep. 1999.
[2] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *IPDPS '13*, 2013.
[3] J. Paudel, O. Tardieu, and J. N. Amaral, "On the merits of distributed work-stealing on selective locality-aware tasks," in *ICPP '13*, 2013.
[4] S. Olivier and J. Prins, "Scalable dynamic load balancing using UPC," in *ICPP '08*, 2008.
[5] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *PPoPP '11*, 2011.
[6] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar, "HabaneroUPC++: A compiler-free PGAS library," in *PGAS '14*, 2014.
[7] "UTS-1.1." [Online]. Available: http://sourceforge.net/p/uts-benchmark/wiki/Home/