J Supercomput (2018) 74:1435–1448
https://doi.org/10.1007/s11227-018-2234-8

Hybrid work stealing of locality-flexible and cancelable tasks for the APGAS library

Jonas Posner[1] · **Claudia Fohry**[1]

Published online: 8 January 2018

Abstract Since large parallel machines are typically clusters of multicore nodes, parallel programs should be able to deal with both shared memory and distributed memory. This paper proposes a hybrid work stealing scheme, which combines the lifeline-based variant of distributed task pools with the node-internal load balancing of Java's Fork/Join framework. We implemented our scheme by extending the APGAS library for Java, which is a branch of the X10 project. APGAS programmers can now spawn locality-flexible tasks with a new `asyncAny` construct. These tasks are transparently mapped to any resource in the overall system, so that the load is balanced over both nodes and cores. Unprocessed `asyncAny`-tasks can also be cancelled. In performance measurements with up to 144 workers on up to 12 nodes, we observed near linear speedups for four benchmarks and a low overhead for cancellation-related bookkeeping.

Keywords Task pool · Work stealing · Task cancellation · APGAS · Java

This paper is an extended version of Jonas Posner and Claudia Fohry: A Combination of Intra- and Inter-Place Work Stealing for the APGAS Library. Parallel Processing and Applied Mathematics Workshops (WLPP), 2017.

✉ Jonas Posner
jonas.posner@uni-kassel.de

Claudia Fohry
fohry@uni-kassel.de

[1] Reseach Group Programming Languages/Methodologies, University of Kassel, Kassel, Germany

1 Introduction

Clusters of multicore nodes are the prevalent architecture in high-performance computing today. To efficiently use them, parallel programs should simultaneously exploit both node-internal shared-memory parallelism and inter-node distributed-memory parallelism.

Hybrid programming is often based on the Partitioned Global Address Space (PGAS) model. It divides the machine into disjointed *places*, such that each place comprises a partition of the global address space and a subset of the computational resources. Every place can access every remote memory partition, but accesses to the local partition are faster. In the Asynchronous PGAS variant of the model, tasks can be created dynamically.

The Asynchronous PGAS model is the basis of the language X10 [7] and the related APGAS library for Java [20]. These systems define an `async` construct to spawn a task on the current place and an `asyncAt` construct to spawn a task on a user-defined remote place.

Various applications deploy locality-flexible tasks that may run on any resource of the overall system. For such tasks, programmers do not want to specify a placement. Instead, it is preferable that the system places them dynamically so as to balance the load. Placement can follow the task pool pattern, in which a large number of tasks are processed by a fixed number of computing resources, called *workers*. The load can be balanced by work stealing, in which an idle worker (called *thief*) takes away tasks from another worker (called *victim*).

The HabaneroUPC++ programming library adopts this concept. It introduces an `asyncAny` construct for spawning locality-flexible tasks and implements it in the runtime system [11]. Unfortunately, the cited publication does not report speedup values and we were not able to obtain speedups with the code provided by the authors [18].

Other Asynchronous PGAS systems do not yet support locality-flexible tasks, but some run-time systems implement intra-place work stealing only. For support of irregular applications, X10 provides a separate Global Load Balancing (GLB) framework in its standard library [24]. A multithreaded variant of GLB for APGAS has been developed in own previous work [17]. Unfortunately, both GLB implementations are restricted to a single worker per place. A multicore system can only be exploited by starting multiple places per node, which causes unnecessary communication and increases the memory load.

This paper introduces a hybrid work stealing scheme, which supports multiple workers per place. For that, we adopted the concept of `asyncAny`-tasks from HabaneroUPC++. However, our scheme is implemented in the APGAS library and internally uses a fundamentally different algorithm: It combines the intra-place load balancing of Java's `ForkJoinPool` [14] with the GLB lifeline scheme for inter-place load balancing. HabaneroUPC++, in contrast, uses the SLAW scheduler [6] for intra-place load balancing and, in inter-place load balancing, selects a suitable victim with the help of network Remote Direct Memory Access (RDMA). Moreover, the HabaneroUPC++ scheme contacts an unlimited number of remote victims, whereas our scheme contacts a fixed number of random victims and lifeline buddies according to the lifeline scheme.

In addition to `asyncAny`, we introduce several related constructs. They include a finish block to wait for the termination of all recursively spawned `asyncAny`-tasks and support for calculating an overall result by reduction from task results. Another new functionality enables the cancellation of `asyncAny`-tasks. Cancellation dequeues all unprocessed tasks that were spawn in the current block. Cancellation is useful for search problems.

We selected the APGAS library for our work, since it is based on the popular Java language, which also gains more and more attention in HPC [2]. Our implementation extends the source code of this library [8], such that APGAS programmers get immediate access to the new constructs. Overall, this paper makes the following contributions:

- It introduces a hybrid work stealing scheme that combines Java's `ForkJoinPool` with GLB's lifeline scheme and allows for task cancellation.
- It describes the implementation of this scheme in the APGAS library for Java.
- It presents experimental results for four benchmarks (Unbalanced Tree Search, Betweenness Centrality, NQueens, and TSP) with up to 144 workers on up to 12 places. Speedups over sequential execution are close to linear in most cases.
- It evaluates variants of NQueens and TSP that cancel the computation after a user-defined solution was found.

The paper is organized as follows. Section 2 starts with background information about APGAS and GLB. Thereafter, Sect. 3 introduces the novel constructs, and demonstrates their usage with examples. The hybrid work stealing scheme and its implementation are described in Sect. 4. Experimental results are presented and discussed in Sect. 5. The paper finishes with related work and conclusions in Sects. 6 and 7, respectively.

2 Background

2.1 APGAS library

The APGAS library brings the parallel programming concepts of X10 to Java [20]. Each place is represented by a single Java Virtual Machine (JVM).

APGAS programmers can spawn lightweight asynchronous tasks that encapsulate computations. At task creation, they explicitly specify an executing place, where the APGAS run-time processes the different tasks with multiple threads. Internally, each place maintains a task pool, which is an instance of the Java class `ForkJoinPool` [14], i.e. workers correspond to Java threads. APGAS users can set the number of workers via `APGAS_THREADS`; default is the number of CPU cores.

In the simplest way, a task is spawned with the `async` construct. This call does not block and always returns immediately. An `async`-task is executed on the current place when a worker becomes available. Java's `ForkJoinPool` assigns tasks in arbitrary order. More generally, a task can be created with the `asyncAt` construct. This construct inserts the task in the `ForkJoinPool` of a remote place, which must be specified as parameter. Like `async`, the `asyncAt` construct returns immediately. Similar to `asyncAt`, the `at` construct sends a task to a specified remote place, where

it is inserted into the local `ForkJoinPool`. Unlike `asyncAt`, this construct blocks until the transferred task has been executed. Finally, the `immediateAsyncAt` construct transfers a task to a specified remote place and then starts a new Java thread there, which immediately executes the transferred task. Such a thread runs concurrently to the local `ForkJoinPool`.

Task creation can be enclosed in a `finish` block. This block's execution ends only when all submitted `async`- and `asyncAt`-tasks, including recursively spawned ones, have been processed.

2.2 Lifeline-based global load balancing

GLB deploys a lifeline scheme, in which each worker runs on a separate place. Each worker maintains its own local task pool, from which it takes tasks to be processed, and into which it inserts any newly generated tasks. GLB uses the following task model:

- Tasks are free of side effects.
- Processing a task generates a task result and possibly new tasks.
- All task results must have the same type and must be reducible with a commutative and associative operator.
- Each worker maintains a partial result and accumulates task results therein.
- The final result is computed from the partial results by reduction.

If a worker runs out of tasks, it tries to steal tasks from another worker. First, the thief contacts up to w random victims and, if not successful, afterwards up to z so-called lifeline buddies. If a victim has no tasks, it responds with a reject message; otherwise, it sends tasks, called *loot*. When all $w + z$ steal attempts have returned unsuccessfully, the thief goes *inactive*. An inactive thief is restarted when a lifeline buddy sends loot in reaction to an earlier recorded steal request. When all workers have become inactive, the final result is computed.

3 Programming with asyncAny-tasks

We incorporated the following novel constructs into the APGAS library and thus made them available to APGAS programmers:

- `asyncAny`: Submits a locality-flexible task. The task is initially placed in the local pool and can later be stolen away to other places.
- `finishAsyncAny`: Suspends until all `asyncAny`-tasks that have been directly or recursively spawned in an associated block have been processed.
- `staticInit`: Creates a copy of static data (e.g. constants) on each place.
- `staticAsyncAny`: Resembles `asyncAny`, but allows to specify an initial placement and expects a list of tasks as parameter.
- `mergeAsyncAny`: Merges its parameter, which is a task result, into the partial result of the local worker.
- `reduceAsyncAny`: Computes the current global result by reduction over the partial results of all workers and returns it.

- `cancelableAsyncAny`: Resembles `asyncAny`, but the submitted task can be cancelled.
- `cancelableStaticAsyncAny`: Combines `cancelableAsyncAny` and `staticAsyncAny`.
- `cancelAllCancelableAsyncAny`: Cancels all unprocessed `cancelableAsyncAny` and `cancelableStaticAsyncAny`-tasks and prohibits spawning new ones.

Listing 1 shows a Hello World example with `asyncAny`. The `finishAsyncAny`-call detects when all `asyncAny`-tasks that are spawn in the loop have been processed. The `asyncAny`-tasks just print "Hello Word" and their place number.

The following Listings 2 and 3 illustrate the use of `asyncAny`-tasks with a complete, compilable code. The example calculates π from integrals. Listing 2 depicts the algorithm, and Listing 3 shows the result class.

For simplicity, all tasks are submitted from place 0. Actually, they are already known at program start, and thus, it would be more efficient to distribute them statically at the beginning. This could be accomplished with `staticAsyncAny`.

In Listing 2, we use a loop (lines 4–11) to start N `asyncAny`-tasks (line 6). Each task produces a task result, which is merged into the partial result of the local worker with `mergeAsyncAny` (line 9). The parameter must be of type `ResultAsyncAny<T>`, which is an abstract APGAS class that contains a member variable `result` of type `T`. Therefore, APGAS programmers must provide an own result class that extends `ResultAsyncAny<T>`. Listing 3 depicts such a class

```
1 finishAsyncAny(() -> {
2   for (int i = 0; i < N; i++) {
3     asyncAny(() -> {
4       System.out.println("Hello from " + here());
5     });
6   }
7 });
```

Listing 1 Submitting N `asyncAny`-tasks in an `finishAsyncAny` block

```
1  final int N = 10000;
2  finishAsyncAny(() -> {
3    final double deltaX = 1.0 / N;
4    for (int i = 0; i < N; i++) {
5      final int _i = i;
6      asyncAny(() -> {
7        double x = (_i + 0.5) * deltaX ;
8        double r = (4.0 / (1 + x * x )) * deltaX;
9        mergeAsyncAny(new PiResult(r));
10     });
11   }
12 });
13 System.out.println("Pi=" +
       reduceAsyncAny().getResult());
```

Listing 2 Calculating π with `asyncAny`

```
1  public class PiResult extends ResultAsyncAny<Double> {
2    public PiResult(double r) { this.result = r; }
3
4    @Override
5    public void mergeAsyncAny(ResultAsyncAny<Double> r) {
6        this.result += r.result();
7    }
8
9    @Override
10   public PiResult clone() { return new
         PiResult(this.result()); }
11 }
```

Listing 3 Result class for π

```
1  finishAsyncAny(() -> {
2      ...
3      cancelableAsyncAny(() -> {
4        ...
5        if (((PiResult)reduceAsyncAny()).getResult() > 3)
             {
6          cancelAllCancelableAsyncAny();
7        }
8      });
9    }
10 });
```

Listing 4 Calculating π with `cancelableAsyncAny`

for our example. As shown, the user must override methods `mergeAsyncAny()` (line 5) and `clone()` (line 10).

When the `finishAsyncAny` call from line 2 of Listing 2 ends in line 12, all `asyncAny`-tasks have been processed and merged their results. Therefore, the final result can be calculated by calling `reduceAsyncAny` (line 13).

Listing 4 extends the π example to demonstrate our novel task cancellation feature. The extension stops the computation as soon as the current sum of task results exceeds a threshold that we artificially set to 3. In Listing 4, tasks are submitted with `cancelableAsyncAny` instead of `asyncAny` (line 3). After each task calculation, `reduceAsyncAny` is called to obtain the current value of the global result (line 5). If this value exceeds the threshold, all unprocessed tasks are cancelled by calling `cancelAllCancelableAsyncAny` (line 6). Obviously, frequent calls to `reduceAsyncAny` slow down the program, and thus, an APGAS programmer needs to control these calls. For instance, the calls can be performed every second or by a single worker only.

4 Design and implementation

We implemented our extensions by modifying the open source code of the APGAS library [8]. The extended code will be published on the first authors homepage.

As mentioned in Sect. 2, APGAS realizes the place internal pools with the Java class `ForkJoinPool`. Consequently, all workers of a place share a single task pool. For the partial results, in contrast, each worker maintains a separate variable. A call to `asyncAny` initially inserts the new task into the local `ForkJoinPool`, just like a call to `async`. However, `asyncAny`-tasks can later be stolen away to other places. For that, we combined the intra-place work stealing scheme of the `ForkJoinPool` class with the lifeline-based global load balancing scheme for inter-place work stealing. The latter is performed by one dedicated management worker per place.

4.1 Management worker

When `finishAsyncAny` is called, one management worker is started on each place. This worker runs in a dedicated Java thread, which is not part of the thread group that executes the `ForkJoinPool`. Listing 5 shows the pseudocode of the management worker's main loop. The worker carries out one loop iteration per second (line 10), except when it is inactive. This approach roughly corresponds to that of GLB, where a main loop iteration is carried out after processing n tasks.

If the number of unprocessed tasks in the local pool falls below the number of local workers (line 3), the management worker starts with work stealing (line 4). Ahead-of-time stealing differs from GLB, where stealing is started only when a worker has no tasks left. On the victim place, steal operations are quick. To avoid that they need to wait, they are performed with `immediateAsyncAt`.

The original lifeline scheme is cooperative, i.e. the thief sends a message to the victim and the victim responds [17]. Our scheme, in contrast, is coordinated, i.e. the thief tries to pull half of the unprocessed tasks out of the victim's internal pool itself. For that, we modified Java's `ForkJoinPool` class to enable pulling out multiple tasks at once. Since the `ForkJoinPool` class deploys internal synchronization, tasks can be removed from the pool concurrently to the running computation of the victim.

If the victim is out of work, the `immediateAsyncAt`-task invokes another `immediateAsyncAt` to notify the thief about the result. If the steal request was a lifeline request, the thief is additionally added to a `ConcurrentLinkedQueue`.

```
1 while (tasks available) {
2   send loot to recorded lifeline thieves;
3   if (not enough local tasks left) {
4     try to steal from up to w+z victims;
5   }
6   if (all local tasks have been executed &&
        all potential victims were contacted) {
7     notify place 0;
8     go inactive;
9   }
10  sleep one second;
11 }
```

Listing 5 Main loop of management worker

This queue needs to be thread-safe, because multiple steal requests can be received and executed concurrently. If the victim has work, the loot is sent to the thief with `(cancelable)staticAsyncAny`. This construct directly inserts the tasks from the loot into the thief's local `ForkJoinPool`.

When $w + z$ steal attempts have returned unsuccessfully, the management worker notifies place 0 (lines 6–9) and changes into inactive state (lines 6–9). Further details are provided in Sect. 4.2. An inactive management worker is reactivated when at least one `asyncAny`-task is inserted into the local `ForkJoinPool`. This may happen through a call to `(cancelable)asyncAny` or `(cancelable)staticAsyncAny`.

4.2 FinishAsyncAny

The existing `finish` implementation performs bookkeeping for every task. This induces a high overhead when the number of tasks is large. Therefore, we implemented a new `finishAsyncAny` construct, which observes only the loot. A call to `finishAsyncAny` starts a new Java thread on place 0, which regularly checks whether (1) the internal pool contains unprocessed tasks, or (2) there are unprocessed tasks in remote pools.

The first condition is checked with standard `ForkJoinPool` methods. For checking the second condition, each place maintains an `int` array `stealCounts` with one entry per place. It is initialized with 0. Before a victim sends loot, it increments its local `stealCounts[thief]`. When a thief receives loot, it decrements its local `stealCounts[thief]`. Just before a management worker goes inactive, it sends its `stealCounts` array to place 0, see line 7 in Listing 5. On place 0, the `stealCounts` arrays are added. If all entries in the sum array are 0, the second condition from above must be met, and all management workers have become inactive. Thus, the `finishAsyncAny`-thread on place 0 terminates all management workers, as well as itself.

4.3 Task cancellation

If tasks are submitted with `asyncAnyCancelable`, they can be cancelled later by calling `cancelAllCancelableAsyncAny`. This call dequeues all unprocessed tasks and prevents new submissions. Since tasks are independent, a cancellation of all unprocessed tasks cannot leave the program in an inconsistent state.

To locate all cancelable tasks, each place maintains a map of type `ConcurrentHashMap<Long, Task>`. The first parameter is a system-wide unique task id. A call of `asyncAnyCancelable` inserts the task into the local `ForkJoinPool` and additionally stores it in the map. Tasks remove themselves from the map as their last operation. When tasks are stolen, they are removed from the victim's map and added to the thief's map.

A call to `cancelAllCancelableAsyncAny` starts an `immediateAsyncAt` on each place. It iterates through all map entries and calls the Java method `cancel()` (from class `ForkJoinTask`) for each task. To prevent new task submissions, each place maintains a boolean flag, which is checked before each task submission by

`asyncAnyCancelable`. The flag is set by the `immediateAsyncAt` and reset at the end of the `finishAsyncAny` block.

5 Experiments

Experiments were conducted on a cluster with 12 homogeneous nodes. Each node has two six-core Intel Xeon E5-2643-v4E5 CPUs and 256 GB of main memory [22]. In a first group of experiments, we measured intra-place speedup by starting a single place on one node and varying the number of workers from 1 to 12. Then, we measured inter-place speedup by varying the number of places from 1 to 12. Here, each place was mapped to a separate node with 12 workers. Java was used in version 9.0.1.

As benchmarks, we adopted Unbalanced Tree Search (UTS) [12], NQueens [5], Betweenness Centrality (BC) [3], and the Travel Salesman Problem (TSP) [1]. UTS counts the number of nodes in a highly irregular tree that is dynamically generated from node descriptors. NQueens calculates the number of placements of N queens on an $N \times N$ chessboard, so that no two queens threaten each other. BC calculates a centrality score for each node of a given graph. TSP determines the shortest roundtrip through a given number of cities.

As baseline for the implementations of UTS, BC and NQueens, we used the code of X10's GLB samples (for UTS and BC) [7] and of HabaneroUPC++'s samples (for TSP) [18], respectively. We ported these codes to Java and therefore used our novel APGAS constructs. The actual calculation logic remained unchanged. For TSP, we wrote an own code, which deploys a parallel branch-and-bound algorithm with heuristics [4]. Each place holds a local optimum, and additionally, there is a global optimum. Whenever a worker discovers a new local optimum, it updates the global optimum and propagates any changes to the other workers.

Moreover, we implemented cancelable variants of NQueens and TSP. Here, the user specifies a limit on the number of NQueens placements and a desired maximal length of the roundtrip, respectively. The programs check once per second if the limit is reached. If so, a call to `cancelAllCancelableAsyncAny` destroys all unprocessed tasks. These variants solve search problems for a sufficiently good solution within a shorter period of time.

UTS, NQueens, and TSP start with a single task, which is submitted with `asyncAny` (or `asyncAnyCancelable`), while the other tasks are spawned dynamically. BC, in contrast, statically initializes all tasks and evenly distributes them over all places with `staticAsyncAny`. The result is a single `long` value for UTS, NQueens, and TSP, and a `long` array with one entry per graph node for BC.

Parameters were set as follow:

- UTS: geometric tree shape, branching factor $b = 4$, random seed $s = 19$, and tree depth $d = 17$. Like in GLB [7], the initial task processes up to 511 tree nodes and afterwards submits a new `asyncAny`-task for half of the remaining tree nodes. Any subsequent task does the same.
- NQueens: number of queens and chessboard size $N = 17$. New `(cancelable) asyncAny`-tasks are spawned only until 11 queens are unplaced, as in the HabaneroUPC++ implementation [18].

– BC: random seed $s = 2$, number of graph nodes $N = 2^{15}$ for intra-place experiments, and $N = 2^{17}$ for inter-place experiments. Since the BC implementation from [7] deploys fine-grained tasks, we combine up to 32 tasks into one `asyncAny`-task, which we observed to perform best.
– TSP: number of *cities* = 25. New `(cancelable)asyncAny`-tasks are only spawned until 15 cities are left for the current path, which we observed to perform best.

Figures 1 and 2 depict the measured speedups for the four benchmarks. In the intra-place case (Fig. 1), all benchmarks achieve near linear speedups. The deviation from linear speedup is up to 13.99% for NQueens, up to 22.78% for UTS (both with 11

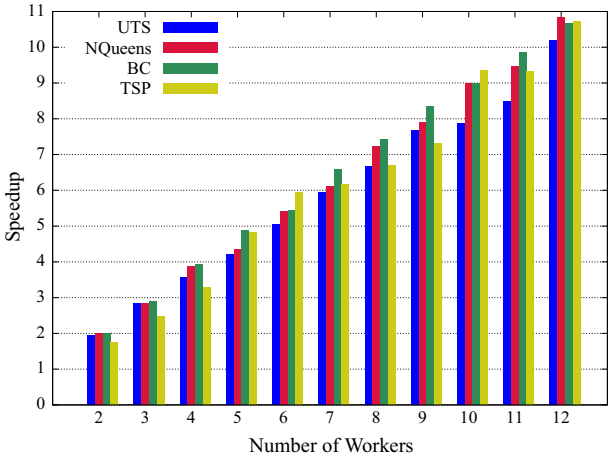

Fig. 1 Intra-place speedups over sequential execution time

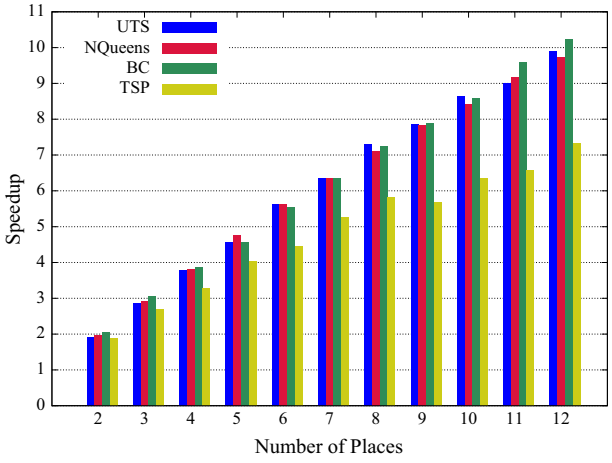

Fig. 2 Inter-place speedups over one place execution time with 12 places

workers), up to 11.12% for BC (with 12 workers), and up to 18.71% for TSP (with 9 workers).

In the inter-place case (Fig. 2), the deviation is a somewhat higher with 14.78% for BC and 18.95% for NQueens (both on 12 places). UTS, on the other hand, has a smaller deviation of 18.27% (on 11 places). The highest deviation was measured for TSP with 40.22% (on 11 places). We expect it to be due to the latency of propagating a new global optimum to an increasing number of workers. The more workers are processing tasks, the more tasks are unnecessarily computed before the new bound becomes effective in the branch-and-bound scheme.

The overall slightly lower performance of inter-place work stealing is probably caused by communication costs. Comparing the hybrid variants with 144 workers to the sequential base variants, UTS achieves a speedup of 100, NQueens of 105, and TSP of 83.

In an additional group of experiments, we started 11 workers per place instead of 12, reserving one CPU core for the management worker. These experiments were run on up to 4 places. We still measured a nearly linear increase in speedup from 11 to 12 workers per place, from which we conclude that the management workers need almost no computational resources. Consequently, it does not pay off to reserve a core for it.

In a final group of experiments, we evaluated our new cancellation functionality. First, we tested the cancelable program variants to make sure that cancellation works properly. As expected, it takes less time to compute a given number of NQueens placements as compared to the total number of placements, and it takes less time to compute an approximate TSP solution as compared to the optimum one.

Second, we estimated the overhead for cancellation management, which includes the internal `asyncAnyCancelable` management of APGAS and the periodic user calls of `reduceAsyncAny`. For these experiments, we configured the cancelable program variants so that they compute all NQueens placements, and the optimum TSP path, respectively. That is, we let them solve the same problem as the non-cancelable variants and compared the respective running times. The overheads of the cancelable variants compared to the non-cancelable variants are depicted in Fig. 3.

As shown in Fig. 3a, the intra-place overhead was at most 6.95% for NQueens and 5.24% for TSP. The inter-place overhead was at most 3.44% for NQueens and 4.21% for TSP. Variations may be due to differences in the task processing order.

6 Related work

As mentioned in Sect. 1, we adopted the concept of locality-flexible `asyncAny`-tasks from HabaneroUPC++ [11]. However, our implementation of hybrid work stealing is fundamentally different. The HabaneroUPC++ scheme does not limit the number of random remote victims, but evaluates them with the help of network Remote Direct Memory Access (RDMA). In contrast, we deploy GLB's lifeline scheme [24] and thus steal from up to $w + z$ remote victims, which are selected without RDMA. Like HabaneroUPC++, we utilize a dedicated management worker for the inter-place work stealing. However, HabaneroUPC++ runs the management worker on a dedicated CPU core that does not participate in the actual computation, whereas we use as many

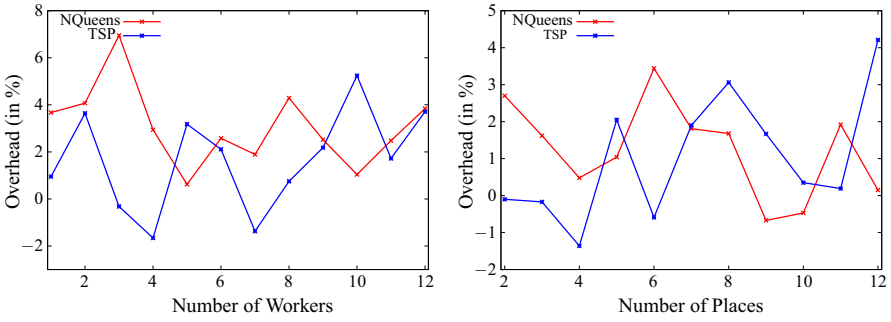

Fig. 3 Intra-place (**a**) and inter-place (**b**) performance overhead of `asyncAnyCancelable` compared to `asyncAny`

computation workers as cores. Finally, HabaneroUPC++ binds to C++, and APGAS to Java.

Yamashita et al. [23] present multistage execution and multithreading for GLB in X10. Each worker maintains an own queue, and each place holds two shared queues for intra- and inter-place work stealing, respectively. However, the overall scheme is quite complicated, and the implementation has problems with network message scheduling.

Paudel et al. [15,16] investigate hybrid task placement in X10 with work stealing and work dealing, respectively. Both papers deploy a dedicated thread for inter-place communication. Programmers use annotations to distinguish tasks into location-sensitive and location-flexible ones. Hybrid work stealing for nested fork/join programs is handled in [9].

A classification and evaluation of task cancellation techniques are given in [10]. Most task-based programming environments do not support task cancellation [21]. An exception is OpenMP [13], where users can cancel parallel regions, sections, loops, and taskgroups. Moreover, HPX [19] supports cancelling individual tasks that have not yet been started or are currently blocked. To the best of our knowledge, our work is the first that provides a cancellation feature for locality-flexible tasks.

7 Conclusions

In this paper, we have presented a combined intra- and inter-place work stealing scheme for the APGAS library. APGAS programmers can submit locality-flexible tasks via `asyncAny` and `asyncAnyCancelable`. These tasks are automatically scheduled by the APGAS run time over all places and their computational resources. Moreover, APGAS programmers can cancel all unprocessed `asyncAnyCancelable`-tasks of a `finishAsyncAny` block.

The paper has described the usage and implementation of the extended APGAS library. Moreover, we ported four benchmarks to this library and observed near linear speedups with up 12 workers on 1 place and a slightly worse speedup with up to 144

workers on up to 12 places. We have also implemented cancelable variants of two benchmarks and observed a management overhead for the cancellation below 7%.

Future work may consider use of RDMA in APGAS, which may further improve the performance of `asyncAny`-tasks. Moreover, cancellation may be extended to running tasks.

Acknowledgements This work is supported by the Deutsche Forschungsgemeinschaft, under Grant FO 1035/5-1.

References

1. Applegate DL, Bixby RE, Chvatal V, Cook WJ (2007) The traveling salesman problem. Princeton University Press, Princeton
2. Diaz J, Munoz-Caro C, Nino A (2012) A survey of parallel programming models and tools in the multi and many-core era. IEEE Trans Parallel Distrib Syst 23:1369–1386. https://doi.org/10.1109/tpds.2011.308
3. Freeman LC (1977) A set of measures of centrality based on betweenness. Sociometry 40(1):35. https://doi.org/10.2307/3033543
4. Gendron B, Crainic TG (1994) Parallel branch-and-branch algorithms: survey and synthesis. Oper Res 42(6):1042–1066. https://doi.org/10.1287/opre.42.6.1042
5. Gik J (1987) Schach und Mathematik. Deutsch Harri GmbH, Frankfurt a. M
6. Guo Y, Zhao J, Cave V, Sarkar V (2010) SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. https://doi.org/10.1145/1693453.1693504
7. IBM: Core implementation of X10 programming language including compiler, runtime, class libraries, sample programs and test suite. https://github.com/x10-lang/x10 (2017)
8. IBM: The APGAS library for fault-tolerant distributed programming in Java 8. https://github.com/x10-lang/apgas (2017)
9. Kestor G, Krishnamoorthy S, Ma W (2017) Localized fault recovery for nested fork-join programs. In: Proceedings of IEEE International Symposium on Parallel and Distributed Processing. https://doi.org/10.1109/ipdps.2017.75
10. Kolesnichenko A, Nanz S, Meyer B (2013) How to cancel a task. Springer, Berlin, pp 61–72. https://doi.org/10.1007/978-3-642-39955-8_6
11. Kumar V, Murthy K, Sarkar V, Zheng Y (2016) Optimized distributed work-stealing. In: Proceedings of Workshop on Irregular Applications: Architectures and Algorithms, pp 74–77. https://doi.org/10.1109/IA3.2016.19
12. Olivier S, Huan J, Liu J, Prins J, Dinan J, Sadayappan P, Tseng CW (2006) UTS: an unbalanced tree search benchmark. In: Languages and Compilers for Parallel Computing, pp 235–250. Springer LNCS 4382. https://doi.org/10.1007/978-3-540-72521-3_18
13. OpenMP ARB: OpenMP specifications. http://www.openmp.org/specifications/ (2017)
14. Oracle: Class ForkJoinPool. https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ForkJoinPool.html (2017)
15. Paudel J, Tardieu O, Amaral JN (2013) Hybrid parallel task placement in X10. In: Proceedings of ACM SIGPLAN Workshop on X10. https://doi.org/10.1145/2481268.2481277
16. Paudel J, Tardieu O, Amaral JN (2013) On the merits of distributed work-stealing on selective locality-aware tasks. In: Proceedings of International Conference on Parallel Processing. https://doi.org/10.1109/icpp.2013.19
17. Posner J, Fohry C (2016) Cooperation versus coordination for lifeline-based global load balancing in APGAS. In: Proceedings of ACM SIGPLAN Workshop on X10. https://doi.org/10.1145/2931028.2931029
18. Rice University: HabaneroUPC++: a Compiler-free PGAS Library. https://github.com/habanero-rice/habanero-upc (2017)
19. STEllAR-GROUP: HPX: The C++ standards library for parallelism and concurrency (2017). https://github.com/STEllAR-GROUP/hpx

20. Tardieu O (2015) The APGAS library: resilient parallel and distributed programming in Java 8. In: Proceedings of ACM SIGPLAN Workshop on X10. https://doi.org/10.1145/2771774.2771780
21. Thoman P, et al (2018) A taxonomy of task-based technologies for high-performance computing. In: Proceedings of International Conference Parallel Processing and Applied Mathematics (To appear)
22. University of Kassel: Scientific data processing. https://www.uni-kassel.de/its-handbuch/en/daten-dienste/wissenschaftliche-datenverarbeitung.html (2017)
23. Yamashita K, Kamada T (2016) Introducing a multithread and multistage mechanism for the global load balancing library of X10. J Inf Process 24(2):416–424. https://doi.org/10.2197/ipsjjip.24.416
24. Zhang W, Tardieu O, Grove D, Herta B, Kamada T, Saraswat V, Takeuchi M (2014) GLB lifeline-based global load balancing library in X10. In: Proceedings of ACM Workshop on Parallel Programming for Analytics Applications. https://doi.org/10.1145/2567634.2567639