Comparison of Load Balancing Schemes for Asynchronous Many-Task Runtimes

Lukas Reitz(✉), Kai Hardenbicker, and Claudia Fohry

Research Group Programming Languages/Methodologies, University of Kassel, Kassel, Germany
{lukas.reitz,fohry}@uni-kassel.de

Abstract. A popular approach to program scalable irregular applications is Asynchronous Many-Task (AMT) Programming. Here, programs define tasks according to task models such as dynamic independent tasks (DIT) or nested fork-join (NFJ). We consider cluster AMTs, in which a runtime system maps the tasks to worker threads in multiple processes.

Thereby, dynamic load balancing can be achieved via work-stealing or work-sharing. A well-performing work-stealing variant is the lifeline scheme. While previous implementations are restricted to single-worker processes, a recent hybrid extension combines the scheme with intra-process work-sharing between multiple workers. The hybrid scheme comes at the price of a higher complexity.

This paper investigates whether this complexity is indispensable by contrasting the scheme with a pure work-stealing extension of the lifeline scheme introduced in the paper. In an experimental comparison based on independent DIT and NFJ implementations and three benchmarks, the pure work-stealing scheme is on a par or even outperforms the hybrid one by up to 3.8%.

Keywords: Work Stealing · Work Sharing · Runtime Systems · Asynchronous Many-Tasking · Task-based Parallel Programming

1 Introduction

Asynchronous Many-Task (AMT) programming, as exemplified by Cilk [2], OpenMP tasks [14], and HPX [7], is a popular approach to tackle irregularity in parallel applications. AMT programs partition the computation into units called *tasks*, and a runtime system (briefly called AMT, as well) maps the tasks to lower-level resources called *workers*. We consider cluster AMTs, for which the workers correspond to threads of multiple processes that may run on different nodes.

AMTs can be classified by their model of task cooperation [6]. In particular, *dynamic tasks* are allowed to spawn child tasks to which their parent task may pass parameters. We consider two subclasses:

R. Wyrzykowski et al. (Eds.): PPAM 2022, LNCS 13827, pp. 14–26, 2023.
https://doi.org/10.1007/978-3-031-30445-3_2

1. *Dynamic Independent Tasks* (DIT) do not communicate, but yield a task result. The final result is calculated from the task results by reduction, e.g., by integer summation. Examples for DIT runtimes include GLB [27] and Blaze-Tasks [15].
2. *Nested Fork-Join* (NFJ) programs begin the computation with one root task. Then each task returns its result to its parent, and the root task yields the final result. Examples for NFJ runtimes include Cilk and Satin [12].

Many AMTs deploy dynamic load balancing, which may be accomplished via *work stealing* or *work sharing*. In work stealing, idle workers (*thieves*) take tasks from other workers (*victims*), whereas in work sharing, busy workers give tasks to others.

A well-performing work stealing variant is Lifeline-based Global Load Balancing, briefly called the *lifeline scheme* [22]. It was first implemented in the Global Load Balancing (GLB) library of the parallel programming language X10 [3] and later ported to Java [17]. Unfortunately, these implementations allow only one worker per process.

A recent *hybrid scheme* overcomes this limitation by combining the lifeline scheme for work stealing between the processes with work sharing among multiple workers within a process. This scheme has been implemented in Java GLB variants for DIT and NFJ, which we denote by $\mathbf{DIT}_{hybrid}$ [5] and $\mathbf{NFJ}_{hybrid}$ [21], respectively.

While the hybrid scheme overcomes the single-worker limitation, its hybrid design has the drawback of a higher complexity. This led us to our research question: Is the complexity of the hybrid scheme indispensable for an efficient extension of the lifeline scheme to multiple workers per process?

To answer the question, we extended the lifeline scheme so that it solely relies on work stealing, but still has multiple workers per process. Our new scheme, which we call *lifeline-pure*, is essentially identical to the lifeline scheme, except that threads instead of processes take over the role of workers. In particular, each worker (thread) within a process maintains its own task queue. When a worker runs empty, it tries to steal tasks from a random worker, which is any thread in the same or in a different process. As will be discussed later, preferring local over global victims may increase the efficiency. Nevertheless, even without such locality optimization in place, we were able to show that the lifeline-pure scheme is on a par or even outperforms the more complicated hybrid scheme by up to 3.8 %.

We conducted our experiments with up to 1280 workers and three benchmarks. Two implementations of the lifeline-pure scheme were used. Both are based on Java GLB, and are called $\mathbf{DIT}_{pure}$ and $\mathbf{NFJ}_{pure}$, respectively. To strengthen our results, the DIT and NFJ implementations were developed independently by the second and first authors of this paper, respectively.

The remainder of the paper is organized as follows. Section 2 provides further details on the load balancing schemes and task models. Then, Sect. 3 discusses the design and implementation of DIT_{pure} and NFJ_{pure}. Experimental results

are presented and discussed in Sect. 4. The paper finishes with related work and conclusions in Sects. 5 and 6, respectively.

2 Background

2.1 Lifeline Scheme

The lifeline scheme [22] deploys *cooperative* work stealing, i.e., thieves ask their victims for tasks, and victims respond by sending tasks or a reject message. When a worker runs out of tasks, it first attempts to steal from up to w random victims. If all random steal attempts fail, it informs z so-called *lifeline buddies*, which are neighbored workers in a connected graph, called the *lifeline graph*. The lifeline buddies record all lifeline steal attempts and possibly answer them later.

Each worker maintains an own local task queue. It takes out tasks for processing and inserts child tasks at one end, and extracts loot for thieves at the other. The workers communicate in work stealing by calling a function on the remote worker, where it is executed by an additional thread. For example, to answer a successful random steal request, the victim calls a function on the thief and passes the tasks as a parameter. The function inserts the tasks into the thief's local task queue, which is synchronized for this purpose.

Listing 1.1 depicts pseudocode for the main loop of each worker. Workers process tasks in chunks of k tasks (line 3), after which they respond to recorded steal requests (line 4). When a worker runs out of tasks, it first tries to steal from random victims (line 6). If all random steal attempts fail, the worker notifies its lifeline buddies and enters an idle state (line 8), from which it can be restarted if a lifeline buddy delivers tasks later.

```
1  do {
2    do {
3      processUpToKTasks();
4      answerStealRequests();
5    } while (tasksAvailable());
6    attemptRandomSteals();
7  } while (tasksAvailable());
8  informBuddiesAndBecomeIdle();
```

Listing 1.1. Main loop of Lifeline-based Global Load Balancing

2.2 Hybrid Scheme

As mentioned in Sect. 1, the hybrid scheme [5] couples the lifeline scheme for work stealing between the processes with work sharing among the workers within a process. It uses two shared queues, which are synchronized to allow accesses from multiple threads:

– an *intra queue* for intra-process work sharing, and
– an *inter queue* chiefly for inter-process work stealing.

Listing 1.2 depicts pseudocode for the main loop of each worker. A process begins with a single worker. After its own spawn, each worker repeatedly tries to spawn additional workers and gives them some tasks (line 4), until some desired maximum number of workers is reached. Then, if one of the shared queues is empty, the worker puts any surplus tasks there (lines 6–11). Afterwards, it processes up to k tasks, and repeats the previous steps as long as it has tasks. When a worker runs out of tasks, it first attempts to take all tasks from the intra queue (lines 14–16), or otherwise from the inter queue (lines 17–19). If both shared queues are empty, the worker shuts down (end of code) and has to be spawned again later.

```
1  do {
2    do {
3      if (numWorkers < numMaxWorkers) {
4        attemptToSpawnAdditionalWorker();
5      }
6      if (intraQueueEmpty) {
7        shareTasksToIntraQueue();
8      }
9      if (interQueueEmpty) {
10       shareTasksToInterQueue();
11     }
12     processUpToKTasks();
13   } while (tasksAvailable());
14   if (!intraQueueEmpty) {
15    takeTasksFromIntraQueue();
16   }
17   if (!tasksAvailable() && !interQueueEmpty) {
18     takeTasksFromInterQueue();
19   }
20 } while (tasksAvailable());
```

Listing 1.2. Main loop of Lifeline-based Global Load Balancing

2.3 Nested Fork-Join and Dynamic Independent Tasks

As already noted, the NFJ and DIT task models deploy dynamic tasks. We assume that the tasks are free of side effects.

For **NFJ**, Listing 1.3 depicts pseudocode of a naive recursive Fibonacci program. The code is invoked on worker 0 by calling `fib(n)`. The `spawn` keyword in line 5 generates a child task and passes `n-1`. The child task calculates `fib(n-1)` recursively. Afterwards, the result is assigned to variable `a` of the parent task. The `sync` keyword pauses the execution of the parent task until all child tasks have returned their results. Thus, the structure of the computation can be regarded as a *task tree*, in which the root task returns the final result.

```
1  fib(n) {
2    if (n < 2) {
3      return 1;
4    }
5    a = spawn fib(n-1);
6    b = fib(n-2);
7    sync;
8    return a + b;
9  }
```

Listing 1.3. Nested fork-join: naive recursive Fibonacci

Work stealing in NFJ$_{hybrid}$ is implemented with the *work-first* policy: When a worker spawns a child task, it puts a description of the parent task into the task queue and branches into the child. The description is called a *continuation* and represents the remaining computation of this task. For instance, the continuation that is generated in line 5 of Listing 1.3 denotes the code in lines 6 to 9 enhanced by the value of n and the knowledge that a will be provided by the child task. The continuation may be processed by the worker itself after having finished the child, or be stolen away. In NFJ$_{hybrid}$, a thief always takes a single task (*steal-one*).

Thus, any work stealing scheme for NFJ must keep track of the parent-child relations and incorporate child results into their parent. We denote these activities as the *fork-join protocol*. The fork-join protocol of NFJ$_{hybrid}$ [21] was adapted from [8] and passes the result of a child task directly to the parent task if the parent is still in the local queue when the child returns. Otherwise, the worker saves the child result in a data structure that is shared between all workers of the process. Saved results are eventually collected as follows: When a task has to wait for its child tasks in a `sync`, this task is sent back to its previous victim. Child results may already reside there, if the child has finished. Otherwise, they are eventually inserted. Since the parent task may have been stolen multiple times, child results may exist on further victims, and the result collection continues there. In contrast to [8], where tasks are returned to their last thief after incorporating all child results, we process them at their first victim.

Unlike NFJ tasks, **DIT** tasks only cooperate through parameter passing from parents to children. Task results are accumulated into worker results, by combining them with a commutative and associative binary operator (e.g., integer summation). Later, each process combines its local worker results to a process result, and finally the process results are combined to the final result.

Listing 1.4 depicts pseudocode of a naive recursive Fibonacci program in DIT. The code is invoked on worker 0 by calling `fib(n)`. Like before, the `spawn` keyword in line 5 generates a task. Method `incrementResult()` adds 1 to the worker result, since `fib(0)` = `fib(1)` = 1. After global termination of all tasks, worker 0 initiates the calculation of the process and final results. Afterwards, the final result may be queried from the system.

Work stealing in DIT$_{hybrid}$ is implemented with the *help-first* policy: When a worker encounters a spawn, it puts the child task into the local task queue

```
1  fib(n) {
2    if (n < 2) {
3      incrementResult();
4    } else {
5      spawn fib(n-1);
6      fib(n-2);
7    }
8  }
```

Listing 1.4. Dynamic independent tasks: naive recursive Fibonacci

and continues to execute the parent task. In DIT_{hybrid}, thieves steal half of the available tasks of a victim (*steal-half*).

3 Design and Implementation of Lifeline-Pure Scheme

The lifeline-pure scheme extends the lifeline scheme with support for multi-worker processes. As noted in Sect. 1, the scheme is essentially identical to the lifeline scheme, except that the workers correspond to threads. Each worker maintains an own local task queue and participates in the work stealing independently of other workers. Also, the lifeline graph and the random victim selection operate at the granularity of workers.

Unlike in NFJ_{hybrid}, we decided to perform all activities of the fork-join protocol separately for each worker within a process in order to reduce contention on the shared data structures. For DIT, as in the hybrid scheme, we first combine the worker results within each process, and then perform a global reduction.

A modification of the lifeline scheme refers to the realization of the communication between a pair of workers. Whereas workers directly communicate with each other in the lifeline scheme, they use a so-called *coordinator* in the lifeline-pure scheme. The coordinator handles all communication, i.e., a worker that is going to send a message to another worker calls a function of its coordinator. The coordinator then sends a message to the remote worker's coordinator. The remote coordinator then forwards the message to the target worker. Global and local worker ids are translated into each other in an obvious way. Figure 1 shows the communication paths, where several workers, denoted as W, communicate with each other through their coordinators, denoted as C@.

Obviously it would be profitable to prefer local over global victims, since process internal stealing has lower communication costs. As of yet, the lifeline-pure scheme does not incorporate such locality optimizations, but the scheme could be easily extended accordingly.

All implementations are based on the "APGAS for Java" library [23], which is a Partitioned Global Address Space (PGAS) platform. We used a modified version of it, which is available in a public git repository [16].

In our implementations, the coordinator is a Java class. Messages between workers of the same process do not get serialized and passed through the network, but are executed in one or more separate threads of Java's fork-join pool.

4 Experimental Evaluation

This section compares the running times of the lifeline-pure and hybrid DIT and NFJ variants, respectively.

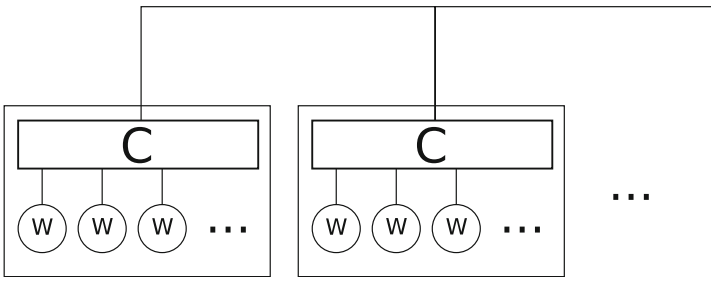

Fig. 1. Communication paths between workers (W) through the coordinator (C) in the lifeline-pure scheme

All experiments were conducted with Java version 17.0.2. We run our programs on the Goethe cluster of the University of Frankfurt [25], where we used a partition of homogeneous Infiniband-connected nodes. Each node is equipped with two 20-core Intel Xeon Skylake Gold 6148 CPUs and 192 GB of main memory. We used up to 32 nodes, with one process per node and one worker per core, resulting in a total of 1280 workers. We report averages over 15 runs.

We used three benchmarks:

- **Fib** (for NFJ): The naive Fibonacci benchmark was presented in Sect. 2. It computes `fib(n)`.
- **UTS**: The Unbalanced Tree Search benchmark dynamically generates a highly-irregular tree and counts its nodes [13]. Users provide a tree depth d, a branching factor b, an initial seed s of a pseudorandom generator, and a probability distribution that determines the tree shape (binomial or geometric).
- **Syn**: The synthetic benchmark counts the nodes of a perfect w-ary tree [19]. Users provide a desired running time T_{calc}, a number m specifies the number of tree nodes per worker, and a task duration variance v as percentage. Each task repeatedly calculates the 5th Fibonacci number recursively until it reaches its task duration. An execution with GLB then takes time $T = T_{\text{calc}} + \epsilon$, where ϵ is the additional time taken by the runtime system, called the runtime system *overhead*. In the case of DIT, ϵ is caused by the load balancing scheme. In the case of NFJ, ϵ is caused by the load balancing scheme and the fork-join protocol.

In all benchmarks, task results are `long` values and the reduction operator is sum.

In preliminary experiments, we found that a so-called *sequential cut-off* reduced the execution times of the NFJ GLB variants significantly: The sequential cut-off c defines a remaining depth (e.g., `fib(n)` calls with $n \leq c$), where the `spawn` statement causes workers to jump into the given function instead of spawning a child task. We implemented a sequential cut-off for Fib and UTS@. Syn did not require one, because the task granularity can be controlled by its benchmark parameters.

Table 1. Benchmark parameters

Benchmark	Parameters
Fib	$n = 67$
	$c = 30$
UTS	$d = 19$
	$b = 4$
	$c = 6$
	geometric tree shape
Syn	$T_{\text{calc}} = 100\,\text{s}$
	$m = 10^6$
	$v = 20\,\%$

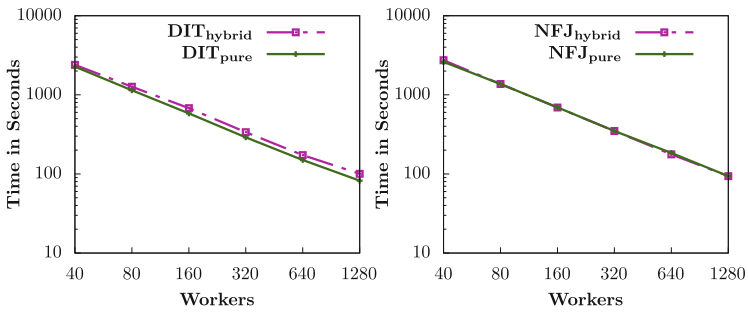

Fig. 2. Strong scaling performance of UTS

For DIT, we used existing implementations of the benchmarks [19]. For NFJ, we slightly improved an existing implementation of Fib [21], and implemented UTS and Syn from scratch.

The used benchmark parameters are shown in Table 1. Recall that both the lifeline-pure and the hybrid scheme process tasks in chunks of k tasks (see Sect. 2). In preliminary experiments, we tried different chunk sizes for each

benchmark and found that the following values for k yield the lowest execution times: $k = 511$ (for UTS in DIT), $k = 16$ (for UTS in NFJ), $k = 10$ (for Fib in NFJ), and $k = 1$ (for Syn).

Figures 2 and 3 show execution times for UTS and Fib. They employ strong scaling to convey an impression of the magnitudes. For each run, we doubled the number of nodes, and thus the number of workers. The measured execution times decrease approximately linearly. Speedups over the execution with one worker are between 1103 and 1243 for 1280 workers.

The strong scaling results for DIT show a bigger difference between the hybrid and the lifeline-pure scheme than those for NFJ@. For DIT, the gap between the two schemes is clearly visible. For NFJ, the gap between the schemes is small and barely visible. This is likely due to the fact, that the used load balancing scheme impacts all the communication in DIT (except the final reduction), but only part of the communication in NFJ (not the fork-join protocol).

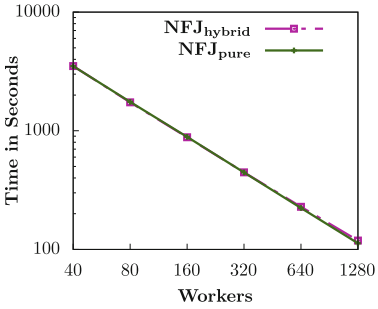

Fig. 3. Strong scaling performance of Fib

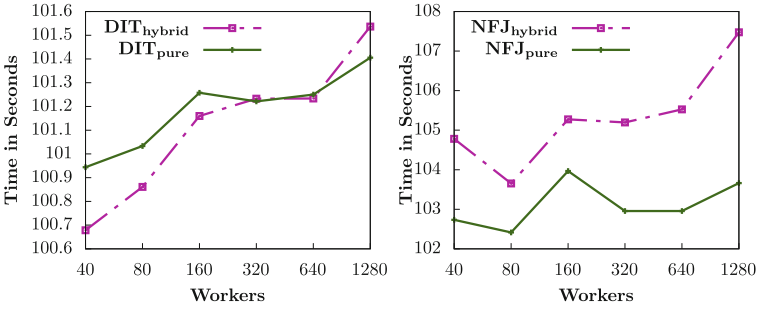

Fig. 4. Weak scaling performance of Syn

Figure 4 shows execution times measured with Syn. We employ weak scaling by keeping T_{calc} constant for all runs. We calculate the overhead as the difference

between the measured execution time and T_{calc}. Since $T_{\text{calc}} = 100\,\text{s}$ for all runs, an execution time of 101 s means, that the runtime system overhead is 1 %. Overheads increase slowly with the number of workers.

For DIT, the overhead is 1.54 % for 1280 workers and the hybrid scheme, and 1.41 % for the lifeline-pure scheme. For NFJ, the overhead is higher than in DIT, since it includes the cost of the fork-join protocol. Because the hybrid and the lifeline-pure scheme both use the same fork-join protocol, we can still compare the overhead. The highest overhead difference between both schemes is about 3.8 % for 1280 workers, where the hybrid scheme has an overhead of about 7.5 %, and the lifeline-pure scheme has an overhead of about 3.7 %.

5 Related Work

AMT, also called task-based parallel programming, goes back until at least the invention of Cilk in the 1990s [2]. Over the years, a variety of AMT programming environments have been proposed, and, especially on shared-memory machines, already found their way into programming practice (e.g., OpenMP tasks [14]).

From a user perspective, major differences between the AMT environments can be seen in their target architectures and task models [6,10,24]. The latter comprise DIT and NFJ, but also several types of dataflow-based, side effect-based, and actor-based coordination. The runtime systems differ in whether they support dynamic load balancing and dynamic task generation, and if they do so, in whether they realize it with work stealing or work sharing.

Work stealing became popular with Cilk [2], but several authors see work sharing on a par or prefer it [4,9]. Both work stealing and work sharing can be implemented in a coordinated way, in which queues are shared between workers, or in a cooperative way, in which they are private. The performance is about the same [1,17]. The work stealing variants also differ in their realization of victim selection and termination detection. Reitz [21] compared different strategies for choosing the number of tasks to be stolen. He used the same NFJ_{hybrid} scheme as we did in this paper.

While the lifeline scheme has traditionally been restricted to single-worker processes, other work stealing variants permit multiple workers. For instance, they combine shared- and distributed-memory work stealing into a two-level algorithm [20], or combine the process-internal load balancing of Java's fork-join pool with the lifeline scheme for inter-process work stealing [18].

These two-level algorithms prefer local over global steals to save communication costs, as do $\mathbf{DIT}_{hybrid}$ and $\mathbf{NFJ}_{hybrid}$. The idea of incorporating locality optimization into work stealing was also applied to hierachical architectures, e.g., [11]. Its usage may further improve the efficiency of our $\mathbf{DIT}_{pure}$ and $\mathbf{NFJ}_{pure}$ schemes.

As mentioned in Sect. 2.3, the fork-join protocol of our NFJ GLB variants was adapted from Kestor et al. [8] where tasks are returned to their last thief after incorporating all child results instead of to their first victim. Similar to the coordinators in the lifeline-pure scheme, they deploy a coordinator per process who communicate with each other by calling functions on remote coordinators.

The first GLB variant for X10 that allows multiple workers per process was presented by Yamashita and Kamada [26]. It was later improved by some tuning mechanism and re-implemented as DIT_{hybrid} in Java [5]. We did not employ the tuning mechanism, since it is irrelevant for our benchmarks.

6 Conclusions

This paper has shown that the lifeline scheme can be efficiently extended to multi-worker processes, without introducing the complexity of a hybrid scheme. Our extension, called lifeline-pure, solely relies on work stealing. We implemented it for DIT and NFJ.

Then we performed an experimental comparison between the lifeline-pure and hybrid schemes for DIT and NFJ, respectively. The experiments were run with three benchmarks and up to 1280 workers on a supercomputer. Even though the lifeline-pure scheme does not use any locality optimizations, we observed it to be on a par or even slightly outperform the hybrid scheme. Interestingly, our results were similar for DIT and NFJ, despite significant differences such as help-first vs. work-first, steal-half vs. steal-one, and the fact that the implementations have been developed independently by different people.

This similarity indicates that our findings may be of a more general nature. In particular, it would be interesting to compare other work stealing variants than the lifeline scheme with hybrid counterparts. Future research should also incorporate locality optimization into the lifeline-pure scheme and quantify the additional performance gain. Moreover, the experiments may be extended to larger benchmarks and other task models.

References

1. Acar, U.A., Charguéraud, A., Rainey, M.: Scheduling parallel programs by work stealing with private deques. SIGPLAN Notices **48**(8), 219–228 (2013). https://doi.org/10.1145/2442516.2442538
2. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999). https://doi.org/10.1145/324133.324234
3. Charles, P., et al.: X10: An object-oriented approach to non-uniform cluster computing. SIGPLAN Notices **40**(10), 519–538 (2005). https://doi.org/10.1145/1103845.1094852
4. Dinan, J., Olivier, S., Sabin, G., Prins, J., Sadayappan, P., Tseng, C.W.: Dynamic load balancing of unbalanced computations using message passing. In: Proceedings of International Parallel and Distributed Processing Symposium (IPDPS), pp. 1–8. IEEE (2007). https://doi.org/10.1109/IPDPS.2007.370581
5. Finnerty, P., Kamada, T., Ohta, C.: Self-adjusting task granularity for global load balancer library on clusters of many-core processors. In: Proceedings of International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM), ACM (2020). https://doi.org/10.1145/3380536.3380539
6. Fohry, C.: An overview of task-based parallel programming models. In: Tutorial at European Network on High-performance Embedded Architecture and Compilation Conference (HiPEAC) (2019)

7. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: HPX: a task based programming model in a global address space. In: Proceedings of International Conference on Partitioned Global Address Space Programming Models (PGAS), pp. 1–11. ACM (2014). https://doi.org/10.1145/2676870.2676883
8. Kestor, G., Krishnamoorthy, S., Ma, W.: Localized fault recovery for nested fork-join programs. In: Proceedings International Symposium on Parallel and Distributed Processing (IPDPS), pp. 397–408. IEEE (2017). https://doi.org/10.1109/ipdps.2017.75
9. Klinkenberg, J., Samfass, P., Bader, M., Terboven, C., Müller, M.: CHAMELEON: reactive load balancing for hybrid MPI+openMP task-parallel applications. J. Parallel Distri. Comput. **138** (2019). https://doi.org/10.1016/j.jpdc.2019.12.005
10. Kulkarni, A., Lumsdaine, A.: A comparative study of asynchronous many-tasking runtimes: Cilk, Charm++, ParalleX and AM++. CoRR abs/1904.00518 (2019). http://arxiv.org/abs/1904.00518
11. Min, S.J., Iancu, C., Yelick, K.: Hierarchical work stealing on manycore clusters. In: Proceedings of the International Conference on Partitioned Global Address Space Programming Models (PGAS), ACM (2011)
12. Nieuwpoort, R.V.V., Wrzesińska, G., Jacobs, C.J.H., Bal, H.E.: Satin: a high-level and efficient grid programming model. Trans. Program. Lang. Syst. (TOPLAS) **32**(3), 1–40 (2010). https://doi.org/10.1145/1709093.1709096
13. Olivier, S., et al.: UTS: an unbalanced tree search benchmark. In: Almási, G., Caşcaval, C., Wu, P. (eds.) LCPC 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72521-3_18
14. OpenMP Architecture Review Board: OpenMP application programming interface (version 5.2). openmp.org (2021)
15. Pirkelbauer, P., Wilson, A., Peterson, C., Dechev, D.: Blaze-tasks: a framework for computing parallel reductions over tasks. Trans. Architect. Code Optim. (TACO) **15**(4) (2019). https://doi.org/10.1145/3293448
16. Posner, J.: Plm-apgas. https://github.com/posnerj/PLM-APGAS
17. Posner, J., Fohry, C.: Cooperation vs. coordination for lifeline-based global load balancing in APGAS. In: Proceedings of SIGPLAN Workshop on X10, pp. 13–17. ACM (2016). https://doi.org/10.1145/2931028.2931029
18. Posner, J., Fohry, C.: Hybrid work stealing of locality-flexible and cancelable tasks for the APGAS library. J. Supercomput. **74**(4), 1435–1448 (2018). https://doi.org/10.1007/s11227-018-2234-8
19. Posner, J., Reitz, L., Fohry, C.: Task-level resilience: checkpointing vs. supervision. Int. J. Netw. Comput. (IJNC) **12**(1), 47–72 (2022). https://doi.org/10.15803/ijnc.12.1_47
20. Ravichandran, K., Lee, S., Pande, S.: Work stealing for multi-core HPC clusters. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011. LNCS, vol. 6852, pp. 205–217. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23400-2_20
21. Reitz, L.: Load balancing policies for nested fork-join. In: Proceedings of International Conference on Cluster Computing (CLUSTER), Extended Abstract, pp. 817–818. IEEE (2021). https://doi.org/10.1109/Cluster48925.2021.00075
22. Saraswat, V.A., Kambadur, P., Kodali, S., Grove, D., Krishnamoorthy, S.: Lifeline-based global load balancing. In: Proceedings of SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 201–212. ACM (2011). https://doi.org/10.1145/1941553.1941582

23. Tardieu, O.: The APGAS library: resilient parallel and distributed programming in java 8. In: Proceedings of SIGPLAN Workshop on X10, pp. 25–26. ACM (2015). https://doi.org/10.1145/2771774.2771780
24. Thoman, P., et al.: A taxonomy of task-based parallel programming technologies for high-performance computing. J. Supercomput. **74**(4), 1422–1434 (2018). https://doi.org/10.1007/s11227-018-2238-4
25. TOP500.org: Goethe-hlr. https://www.top500.org/system/179588
26. Yamashita, K., Kamada, T.: Introducing a multithread and multistage mechanism for the global load balancing library of X10. J. Inf. Process. **24**(2), 416–424 (2016). https://doi.org/10.2197/ipsjjip.24.416
27. Zhang, W., et al.: GLB: lifeline-based global load balancing library in X10. In: Proceedings of Workshop on Parallel Programming for Analytics Applications (PPAA), pp. 31–40. ACM (2014). https://doi.org/10.1145/2567634.2567639