DOI 10.12694/scpe.v16i2.1088
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 16, Number 2, pp. 169–185. http://www.scpe.org
© 2015 SCPE

FAULT TOLERANCE SCHEMES FOR GLOBAL LOAD BALANCING IN X10

CLAUDIA FOHRY, MARCO BUNGART, AND JONAS POSNER *

Abstract. Scalability postulates fault tolerance to be efficient. One approach handles permanent node failures at user level. It is supported by Resilient X10, a Partitioned Global Address Space language that throws an exception when a place fails.

We consider task pools, which are a widely used pattern for load balancing of irregular applications, and refer to the variant that is implemented in the Global Load Balancing framework GLB of X10. Here, each worker maintains a private pool and supports cooperative work stealing. Victim selection and termination detection follow the lifeline scheme. Tasks may generate new tasks dynamically, are free of side-effects, and their results are combined by reduction. We consider a single worker per node, and assume that failures are rare and uncorrelated.

The paper introduces two fault tolerance schemes. Both are based on regular backups of the local task pool contents, which are written to the main memory of another worker and updated in the event of stealing. The first scheme mainly relies on synchronous communication. The second scheme deploys asynchronous communication, and significantly improves on the first scheme's efficiency and robustness.

Both schemes have been implemented by extending the GLB source code. Experiments were run with the Unbalanced Tree Search (UTS) and Betweenness Centrality benchmarks. For UTS on 128 nodes, for instance, we observed an overhead of about 81% with the synchronous scheme and about 7% with the asynchronous scheme. The protocol overhead for a place failure was negligible.

Key words: Resilient X10, task pool, GLB, algorithmic resilience, lifeline scheme

AMS subject classifications. 68M15, 68W10, 68Q10

1. Introduction.

Large-scale applications are likely to encounter hardware failures during their execution. Consequently, fault tolerance is of crucial importance. Checkpoint/restart is an established approach, but application-specific techniques may induce less overhead.

Resilient X10 [4] provides an interesting platform to experiment with user-level fault tolerance. This extension of the language X10 raises an exception in the event of a permanent place failure. Moreover, Resilient X10 provides an inquiry function to check a particular place's liveness. Failure notification to programs is quite a unique feature in current parallel programming systems [5].

The base language X10 [3] follows the Partitioned Global Address Space (PGAS) model, and provides a shared memory abstraction on top of distributed hardware. Nodes of a compute cluster are modeled as places, which comprise a set of processors and a memory partition. Access to local memory is faster than access to remote memory, and the difference is visible to the programmer.

On the algorithm side, we consider task pools, which are a widely used pattern for load balancing of irregular applications. Moreover, tasks are deployed by several modern parallel programming systems, where they replace or complement threads/ processes as a central construct for specifying parallelism. Examples include Cilk [6], OpenMP [25], Chapel [7], and X10 [3].

The use of tasks is promoted for reasons such as ease of programming and load balancing support. Beyond that, tasks provide the additional benefit of easy migration. Since tasks do not refer to a thread / process number in their code, they may be moved away from a faulty place, without having arranged for that in the application code.

Despite their importance, fault-tolerant task pools have received little attention in previous research. In contrast, much work has been conducted on fault tolerance for the related master/worker and parallel divide-and-conquer patterns. The former was chiefly considered in MapReduce systems such as Hadoop [8]. Unlike task pools, Hadoop considers the set of tasks to be fixed from the beginning, and a central master has an overview of all tasks. In divide-and-conquer algorithms, the overall result is computed along the call tree, which is traversed bottom-up after the computation of leaves. Consequently, children must preserve a link to their parents, and parents can recompute their children by need. Fault-tolerant divide-and-conquer algorithms also

*Research Group Programming Languages / Methodologies, University of Kassel, Kassel, Germany contact: {fohry@ | marco.bungart@ | jonas.posner@student.}uni-kassel.de

handle the case that subtasks are stolen away recursively [9, 10, 11]. The schemes perform well for divide-and-conquer, but would induce too much overhead in our setting. A third group of related work specifically deals with idempotence when recomputing tasks with side effects (e.g. [20]).

Unlike related work, we assume that tasks may generate other tasks dynamically, are free of side effects, and produce results that are combined by reduction. This task model is used by various search and optimization algorithms, as represented by the Unbalanced Tree Search [12] and Betweenness Centrality [13] benchmarks. It also underlies the Global Load Balancing framework GLB, which is part of the X10 standard library.

In this paper, we consider the particular type of task pool that is implemented in GLB. It stores tasks in a collection of private pools, each of which belongs to a single worker. A worker operates on its own pool most of the time. Only when this pool is empty, it asks a coworker for tasks. Victim selection and termination detection follow an efficient state-of-the-art scheme, called the lifeline algorithm [1].

We introduce two different fault tolerance schemes that extend GLB. Both regularly save the local task pool contents of one worker in the main memory of another. When a place fails, its backup partner takes over the lost tasks. In case of unfavourably correlated failures, such as simultaneous loss of a place and its backup partner, the program aborts with an error message. Successful completion, on the other hand, guarantees that the final result is correct, despite possible failures.

In both schemes, particular attention was paid to stealing, to avoid inconsistencies between victim and thief after a place failure. The first scheme, which has been introduced in [2], defines a conservative steal protocol. It mainly adopts synchronous communication and cautiously aborts the program in any potentially critical situation.

The second scheme improves on the first one wrt. efficiency and robustness, where robustness characterizes the number of situations that lead to program abort. The improvements have been achieved by a major redesign. Most importantly, the second scheme relies on asynchronous communication consistently. Thus, a worker may continue processing tasks while waiting for an outstanding communication reply. To keep complexity manageable, an actor-like communication structure has been devised. Moreover, the second scheme exploits the redundancy that is immanent in stealing. As compared to the first scheme, the second scheme furthermore reduces handshaking in the steal protocol, combines multiple steal requests in a transaction, and adopts several more minor changes.

In addition to the two schemes presented in this paper, we are currently working at a third scheme. That scheme may be even more efficient, but at the price of reduced flexibility. In particular, it can only be applied to task pools that are organized as a collection of stacks. The UTS benchmark considered in this paper, for instance, can not be handled with the third scheme. The second and third schemes share the idea of an actor-like communication structure. The third scheme has been outlined in [15], with focus on X10 language support, but its implementation is still ongoing.

The schemes described in this paper have been implemented in X10, by extending the GLB source code. Our own code which can be obtained for free from the second author's homepage.

Apart from the unfavourably correlated failures mentioned above, the two schemes can cope with any number of permanent place failures. To avoid unfavourable correlations, we allow only one worker per cluster node. This assumption partly stems from GLB, which permits one worker per PGAS place, but goes farther by requiring that only one place is mapped to each node.

Experiments have been run with the UTS and BC benchmarks. While we observed significant overheads with the first fault tolerance scheme, the second scheme was about as fast as the original GLB. The overhead for restore was low in all cases.

The paper starts with background on X10, task pools, and the GLB library in Sect. 2. Then, Sect. 3 introduces some basic concepts that are common to both fault tolerance schemes, and gives an overview of their respective approaches. Details of the synchronous scheme are presented in Sect. 4, and details of the asynchronous scheme in Sect. 5, respectively. Sect. 6 describes our experimental setting, reports performance numbers, and discusses results. Finally, Sects. 7 and 8 are devoted to related work and conclusions, respectively.

2. Background.

2.1. X10 and Resilient X10. X10 is a novel parallel language from IBM [3], which supports object orientation and exception handling in a similar way as Java. Following the Asynchronous PGAS (APGAS)

programming model, gives the programmer the view of a shared address space that is divided into disjoint partitions. The notion of *place* captures a memory partition and a set of processors, which have faster access to the local memory partition than to remote data.

The placement of data and computations is controlled by the programmer. Access to remote data requires to move the computation to the remote place. This is accomplished with the `at` keyword. In a place change, part of the data from the original place is transparently copied along. Parallelism is specified orthogonally. Using the `async` keyword, a programmer starts an asynchronous task on the current place. Tasks are called *activities*, and can be moved to a remote place by combining `async` and `at`.

Later in this paper, we will refer to remote computations launched via `at` or `at async` as messages. This notion reflects that, in both cases, a computation request, possibly accompanied by data, is sent to the remote place. There, it is queued until a remote thread is available for its execution. Messages sent with `at` are called *synchronous*, since the activity at the origin place is suspended until the remote activity returns, possibly with a result. Messages sent with `at async` are called *asynchronous*, since the activity at the origin place immediately continues with its sequential flow of control.

In asynchronous communication, the origin activity is not notified on completion. Spawning of activities can, however, be enclosed in a `finish` block. At the end of the block, the parent activity waits until all spawned activities and their descendants have terminated.

For place-internal load balancing, a work-stealing scheduler is built into the X10 runtime system. It assigns the activities to the available threads. This scheduler can not be used for global load balancing, since the assignment of tasks to places is under programmer control. Therefore, it is complemented by GLB, which belongs to the X10 library since version 2.4.2. GLB tasks are more heavy-weight than X10 activities. For their deployment, a GLB user must implement interfaces in a custom class.

Since version 2.4.1 of end-2013, X10 supports resilience in its runtime system. It is switched on by setting some environment variables. We deployed resilience mode 1, which has the limitation that place 0 must not fail, or otherwise the program aborts with an error message.

Resilient X10 provides two mechanisms for failure notification. First, a `DeadPlaceException` (DPE) is raised in the event of a failure. When a place fails before or during synchronous communication, the DPE is delivered to all parent activities instead of the regular reply. In asynchronous communication, the DPE is delivered to parent activities, as well, but usually they catch the exception only at the end of a surrounding `finish` block. Second, X10 provides an inquiry function `isDead(...)`, which may be called by any place to check any other particular place's liveness.

2.2. Task Pools and GLB. The paper refers to the particular type of task pool that is used by GLB. Tasks are assumed to be free of side effects, and may generate other tasks dynamically. The overall result is computed by reduction from individual results of each task.

The GLB task pool comprises a distributed data structure and a set of workers. The data structure is a collection of private pools, each of which belongs to a single worker and is stored at the worker's place. Initially, one or several workers may have tasks in their pool, which is referred to as dynamic or static initialization, respectively.

Workers are realized by activities. Each worker runs a loop, in which it repeatedly takes a task out of the local pool, processes it, and possibly inserts new tasks generated. When the local pool is empty, the worker contacts one or several coworkers, called victims, and asks them for tasks.

Victim selection and termination detection follow the lifeline scheme [1]. According to this scheme, a worker successively contacts up to w random coworkers and z lifeline buddies. The latter in their entirety form an appropriate graph. If a victim has no tasks to share, it rejects the request and, if it is a lifeline buddy, additionally stores it. If a lifeline buddy obtains tasks later, it shares them with the stored worker.

When all $w + z$ steal attempts failed, the worker activity ends. If a lifeline buddy sends tasks later, it restarts the worker by spawning a new activity at the corresponding place. Note that multiple lifeline buddies may have stored steal requests from our worker simultaneously, and may, thus, send tasks at the same time later. While only the first message arriving leads to restart, the worker may consequently receive tasks when its pool is non-empty. Similarly, it may receive tasks from a lifeline buddy while waiting for the answer to an outstanding steal request.

The computation terminates when all workers have ended. Then, the final result is computed by reduction from partial results, which are collected and combined by each worker during task processing. Termination is detected by an outer `finish` block, which surrounds all starts and restarts of worker activities.

GLB assumes that there is only one worker per place. Additionally, it precludes any parallel activities at this place. This is enforced by setting environment variables. The assumption is quite restrictive, but eliminates any need for place-internal synchronization. Thus, a worker may, without distruption, alternatingly process up to n tasks, receive incoming messages, and invoke local functions to answer the received steal requests. Recall that messages correspond to remote activities. In GLB, these activities are queued, but are not allowed to run until the worker calls `Runtime.probe()` to receive the messages. This call releases the worker's thread, and all pending activities are scheduled sequentially in any order. GLB uses the following types of messages:

- `give`: The worker receives tasks from a victim. This message may be the response to an outstanding steal request, or originate from an older request stored by a lifeline buddy. In either case, the corresponding activity integrates the tasks into the worker's pool by calling a `merge` function. If the worker has ended, it is additionally restarted.
- `trySteal`: The worker is the victim of a steal request. If its pool is empty, the corresponding activity immediately replies with a `noTasks` message. Otherwise, it records the request at the worker's place.
- `noTasks`: This is the rejection of a former steal request.

When all messages have been received, the worker activity resumes and sends out tasks to the thiefs recorded. The first thief gets half of the tasks, the second a quarter, and so on. Any remaining thieves are sent a `noTasks` message. Since the requests are taken from a data structure, the worker is able to prefer random thieves over lifeline thieves.

Steal requests are sent asynchronously with `at async`, i.e., the thief spawns a remote activity and continues. So it remains responsive to requests from others. The victim sends its reply asynchronously, as well. Request and reply are related by a volatile variable, which is located at the thief place. The thief sets this variable when sending out the request, and the victim resets it later.

From a user's perspective, GLB defines a class and two interfaces that must be implemented. In particular, a GLB user must define a data structure and the following access functions for the private pools:

- `merge` and `split` integrate tasks and split the pool, respectively.
- `process(n)` takes n tasks out, processes them, and inserts any newly generated tasks. If less than n tasks were available, the function returns `false`.

Note that GLB does not restrict the data structure, except that the functions must be provided. While the fault tolerance scheme in [15] only allows stacks, the schemes introduced in the present paper do not restrict GLB's flexibility. As a minor restriction, we forbid use of the `yield()` function, which may be invoked by GLB user code to interrupt long-running tasks. A workaround is explained in Sect. 6.

3. General Structure of the Fault Tolerance Schemes. Our schemes handle the following issues:

- writing regular backups and maintaining a ring structure of backup places,
- recognizing node failure,
- keeping backups consistent during stealing, and
- returning to a consistent state after failures.

The first and second issues are handled in a similar way by the two schemes, and are discussed in this section. The third and forth issues are handled differently with quite complex protocols. Sect. 4 provides the details for the synchronous scheme, and Sect. 5 for the asynchronous one.

Scope of Fault Tolerance. Fault tolerance begins with writing the first backup right after task pool initialization. If a failure occurs earlier, the program can be restarted without notable loss of time.

The number of cases that lead to program abort is lower for the asynchronous than for the synchronous scheme, as captured by the term robustness. Both schemes guarantee that the program either outputs a correct result, or crashes.

Any fault tolerance scheme must find a compromise between overhead and robustness. Our schemes strive to hold each relevant data element at exactly two locations at any time. If the two locations fail simultaneously, the program is aborted. While the redundancy level could in principle be changed from two to some other value, a fair balance of redundancy among data elements is desirable to minimize the backup overhead for a

given level of robustness. The asynchronous scheme gets closer to a fair balance than the synchronous scheme, as explained in Sect. 5.1.

Our algorithms suppose a reliable communication layer, i.e., if both sender and receiver of a message are alive, the message must eventually be delivered. A timing guarantee is not needed, though. In particular, different messages from the same sender to the same receiver may overtake. This may happen since, at `Runtime.probe()`, pending activities are scheduled in any order.

Ring Structure. Let P denote the number of places. In both schemes, each worker regularly writes a backup to the main memory of another place. For that, workers are arranged in a ring, and numbered $0 \dots P-1$. Outside failures, worker i regularly writes its backup to the main memory of worker $(i+1) \bmod P$, which is called its backup partner `Back`(i). Vice versa, if `Back`(i)$= j$, we denote the predecessor as $i =$`Forth`(j).

In case of failures, the gap is bridged, i.e., the next node alive along the ring takes the role of `Back`(i). Re-establishing the ring structure is accomplished by a restore protocol. These protocols differ between the synchronous and asynchronous schemes and are explained in Sects. 4 and 5.

Logs. In both fault tolerance schemes, each worker keeps a log of the places that it has taken over due to restore. The relation of taking over is transitive, i.e., if worker $i-2$ restored $i-1$, and later i restored $i-1$, then we say that worker i has taken over both workers $i-2$ and $i-1$.

Logs are inspected when a worker receives a restore request and must decide whether the program can recover. Consider, for instance, the sequence of workers $i-2, i-1, i$. If $i-1$ fails first, and $i-2$ later, then i can perform $i-2$'s restore if and only if it has $i-1$ in its log and `Forth`(i) $= i-2$.

The synchronous scheme stores logs as a list of all workers restored. The asynchronous scheme, instead, only stores `Forth` as its log, relying on the invariant that each worker i must have taken over all workers [`Forth` $+ 1 \dots i-1$] (cyclically), or otherwise `Forth` is invalid. There must not be gaps between `Forth` and i, since any missing worker's data would be lost. In both schemes, the log is part of a worker's backup data.

Backup Writing. Regular backups are written every kn processing steps, called a *backup interval*. Here, n denotes the GLB parameter, and $k \geq 1$ is an additional parameter. To determine the length of a backup interval, each worker counts steps independently and, at the end, autonomously sends its backup. The backup comprises the current contents of the local task pool, the current value of the partial result, as well as the worker's log. These data are copied to a `val` variable and transparently sent by X10's `at` construct.

Our algorithm writes backups only right before a `process(n)` call. At these moments, the task pool contents, supplemented by the current value of the partial result, defines a worker's state in full.

Shadowing Results of partially finished Tasks. Sometimes, the execution of a single task takes too long to keep a worker responsive. Therefore, GLB defines a `yield()` method. A user can invoke this method inside `process(n)` to interrupt task execution and call `Runtime.probe()`. Our fault tolerance schemes do not support `yield()` for two reasons: First, `yield()` would complicate the program structure, especially for the asynchronous scheme, by introducing an additional `Runtime.probe()` call. Second, at a `yield()` call, a worker's state is not fully represented by the task pool contents.

One of our benchmarks (Betweenness Centrality) deploys long-running tasks, so we still had to find a workaround. Within the execution of a single task, BC performs a breadth-first search on a graph to find all shortest paths from one specific node to all others. Since the number of nodes is typically large, `yield()` should be called in-between. An alternative is provided by one of the BC sample codes that are supplied with GLB. This code does not use `yield()`, but instead decomposes each `process(n)` into several `step` calls.

After each `step` call, the worker may suspend to answer steal requests. To continue thereafter, it needs to save its internal state. This state is larger than the backup state, we denote it as *shadow*. The shadow is only visible to the particular worker, not to others. During each `process(n)`, the worker operates on the shadow, and at the end of `process(n)`, the shadow's result is combined into the partial result. When a backup is written during `process(n)`, it covers the state at the beginning of `process(n)`, but not the shadow. Maintenance of shadows induces additional overhead for copying after each `step`.

Communication Structure. As noted before, the major difference between our two fault tolerance schemes concerns communication. While the synchronous scheme uses X10's `at` construct in most cases, the

asynchronous scheme uses `at async` everywhere.

Asynchronous communication improves efficiency, since a worker may continue processing tasks while communicating with others. Moreover, it remains responsive this way.

On the backside, asynchrony complicates the program structure, since a worker may have to manage multiple outstanding requests at the same time. To reduce complexity, we designed an actor-like communication structure, in which a worker alternatingly processes tasks, receives messages, and carries out the actions required by these messages. The structure is inspired by GLB, but goes farther by consistently applying this three-phase structure to all types of communication. This allows us to process messages in a well-defined order, and to prioritize messages. The scheme resembles the actor model [16] insofar as a worker, except for processing tasks, is a passive entity, and only becomes active upon message receipt.

Failure Notification. As noted in Sect. 2.1, X10 supports failure notification by DPEs and by the `isDead(...)` function. DPEs are well-suited for the synchronous scheme. Throughout the corresponding program, for timely failure notification, all place changes are enclosed in `try-catch` blocks. When a place fails during a regular backup, its predecessor is informed immediately and can initiate the restore protocol (see Sect. 4.2). When a place fails during stealing, the DPE is received by a different worker, which broadcasts it to all other workers. The broadcast is an expensive, but timely way to reach all workers that are currently involved in a communication with the failed worker and whose identities may not be known to the sender.

Unfortunately, DPEs are not suited for the asynchronous scheme, since they are only caught by the outer `finish`, which comes too late. Therefore, we do not use DPEs in the asynchronous scheme, but instead rely on `isDead(...)`. This X10 issue is further discussed in [15].

We implemented a monitoring scheme, in which each place regularly inquires its backup place's liveness by calling `isDead(...)`. Monitoring is disturbed if a worker has ended and thus can not take the initiative for the regular calls. Here, monitoring deploys so-called *ghost activities*, which are temporal re-activations of an ended worker.

To state it in more detail, a worker regularly calls `isDead(...)`, as noted. If the successor is dead, the restore protocol is invoked. If it is alive, the worker sends a `monitor` message. The corresponding remote activity checks whether the successor has ended and, if so, starts a ghost activity. The ghost activity is responsible for 1) invoking `isDead(...)` on the successor's successor, and 2) calling `monitor` recursively, if needed. It can be easily verified, that any failure is recognized this way.

The scheme is speeded up by a second failure notification mechanism, called *timeouts*. Timeouts are deployed in the asynchronous protocols: When a worker has sent a message, it typically expects a reply, even though it does not explicitly wait for it in the actor scheme. These replies are stored in a list, together with some time limit. Occasionally, the worker runs through the list. For all replies that have exceeded their time limit, it invokes a synchronous communication to the respective communication partner. If it is dead, it takes appropriate action according to the respective protocol. In addition, it informs the place's `Forth` who will initiate the restore protocol.

4. Synchronous Fault Tolerance Scheme. While the basic approach of our fault tolerance schemes has been discussed in Sect. 3, steal and restore are more complex. Since the approaches in the synchronous and asynchronous schemes are fundamentally different, we handle the synchronous approach in this section, and the asynchronous approach in Sect. 5.

4.1. Steal Protocol. Stealing requires special arrangements to avoid inconsistencies between thief place `F`, victim place `V`, as well as their backups at `Back(F)` and `Back(V)`. Otherwise tasks may be computed twice or not at all. The protocol is conservative in that the program is cautiously killed in any possibly inconsistent situation, to guarantee that a computed result is always correct.

Figure 4.1 depicts the steal protocol, which was introduced in [2], with time advancing from top to bottom. In the figure, a wavy line indicates that the place is processing tasks (if available), and a solid line marks actions that are part of the protocol. At the backup places, wavy lines are omitted for clarity before and after their involvement. A thick dotted line represents the queueing of a request, i.e., the request has to wait until the worker calls `Runtime.probe()`. An asterisk denotes an asynchronous message sent via `at async`, whereas the other arrows correspond to synchronous communication via `at`. To state it in more detail, `V` invokes three `at`s:

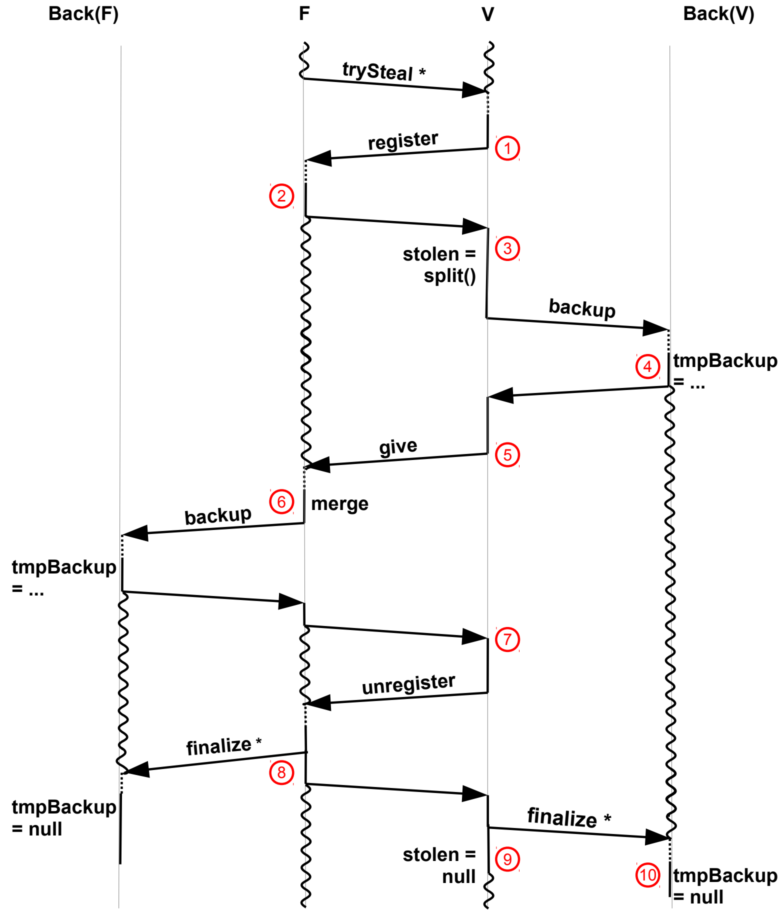

Fig. 4.1. *Steal Protocol of the Synchronous Scheme*

from ① to ③, from ⑤ to ⑦, and from about ⑦ to about ⑨. During the second `at`'s operation at `F`, another `at` to `Back(F)` is launched.

The protocol starts with a `trySteal` request. If `V` has no work, it responds as explained in Sect. 2.2, and no further action is taken. Otherwise, `V` registers at place `F`. From ② to ⑧, `F` is not allowed to accept other register requests.

After registration, `V` splits the pool by calling the user-provided `split` function. Then, it saves the tasks to be stolen and writes a backup without them. While `tmpBackup` $\neq$ `null`, there may be an inconsistency, since `Back(V)` does not know whether `F` and `Back(F)` have already taken over the tasks.

After the backup, `V` sends the tasks to `F`, called `give` in the figure. On arrival, `F` merges the tasks into its own pool. The pool is not necessarily empty, since section ② to ⑧ of the protocol may be repeated if different lifeline buddies send work at about the same time.

Analogously to `V`, thereafter `F` writes a backup and starts processing the tasks just received. The remaining arrows display some handshaking to let all places know when the protocol has finished. Then, variables `tmpBackup` and `stolen` may be reset, and `F` and `V` may accept other requests.

Looking back, there are two situations in which backups are written: regularly every kn steps, and during stealing. To reduce overheads, we restart the kn period after each steal-related backup. Moreover, a place first checks for steal requests before writing a regular backup.

The protocol in Fig. 4.1 considers the base case of a single steal request. Other cases are reduced to this

one as follows:

- If `V` receives other steal requests between ① and ⑨, they are rejected or delayed until ⑨.
- If `V` receives a `register` from `F` between ① and ③, a place number-based ordering guarantees that only one of the crossing `register`s between `F` and `V` passes. The other is delayed until ⑨. Crossing `register`s may occur when `F` is `V`'s lifeline partner and wants to answer a steal request from `V` that it has recorded long ago.
- If `V` receives other register requests between ① and ⑨, they are delayed until ⑨
- If `F` receives `trySteal`s or `register`s between ② and ⑧, they are rejected or delayed until ⑧.
- If `F` receives other `register`s before ②, section ② to ⑧ is run for the first request arriving, and the others are delayed until ⑧. Hence, for "our" request, the protocol corresponds to the base case, except for a larger delay between ① and ③ in this rare case.
- At `F` and `V`, backup requests from `Forth(F)` and `Forth(V)` are processed as usual, concurrently to the steal protocol.
- During the protocol, regular backups of `F` and `V` are skipped and replaced by the steal backups.
- If places `F` and `Back(V)`, or places `V` and `Back(F)` coincide, the respective roles are pursued concurrently.

4.2. Restore Protocol. The restore protocol is depicted in Fig. 4.2. After `Forth(P)` has been notified of the failure, as explained in Sect. 3, it locates `Back(P)` as being the next place alive in the ring. Then, `Forth(P)` sends a `restore(P)` message to `Back(P)`. Upon receipt, `Back(P)` inspects its log, as further explained below. If recovery is possible, `Back(P)` merges the tasks from the backup into its own local pool, combines the partial results, and inserts `P` into its log. Thereafter, it sends a backup of its new state to `Back(Back(P))`. This backup is called a taken-over backup, or shortly `TO-backup`.

After the `TO-backup`, `Back(P)` reports completion to `Forth(P)`, which is denoted by `RSTack` in the figure. Thereupon, `Forth(P)` sends a backup to `Back(P)`. It is called an inauguration backup, or shortly `IA-backup`, since `Back(P)` is `Forth(P)`'s new backup place. Note that the protocol uses synchronous communication, therefore the arrows labelled `RSTack` and `IAack` correspond to implicit returns from `at` calls.

To explain the usage of the log, consider the case that `Back(P)` fails at (R3). Although `Forth(P)` recognizes `Back(P)`'s failure, it has no clue when the failure occured. Therefore, `Forth(P)` looks up `Back(Back(P))` and requests `P`(!)'s backup, as it would do if `Back(P)` died long ago. In our setting, `Back(Back(P))` has received `Back(P)`'s backup, including `P`'s data, as indicated by the log. Therefore, `Back(Back(P))` transforms the request to restore `P` into a request to restore `Back(P)`, and, on completion, reports success to `Forth(P)`. Because of the ring structure, the scheme extends to any number of failed places.

If `Back(P)` fails before (R2), `P` is not contained in the log, and therefore `Back(Back(P))` aborts the program. Between (C1) and (C2), `Forth(P)`'s data are unsecured. Thus, if `Forth(P)` dies during this time, `Forth(Forth(P))` will neither find `Forth(P)`'s backup place, nor a place that has `Forth(P)` in its log. This is recognized by the first place alive, which aborts the program.

4.3. Recovery and Correctness. The following items show that, during the steal protocol, a failure of `F` and/or `V` can either be recovered successfully, or halts the program:

- Failure of `F` before sending the ② → ③ message: Because of the register call, `V` is notified of the failure. Since the stealing has not yet begun, recovery corresponds to the base case.
- Failure of `F` after ② but before sending the message into ⑦: `V` re-merges the stolen tasks into its queue and directs `Back(V)` to reset `tmpBackup`. Depending on the state of `F`'s backup, `Back(F)` either restores the old backup of `F`, or kills the program. When the program goes on, all places consistently see the tasks in question at `V`.
- Failure of `F` right before or during `V`'s `unregister` call: `V` continues with the protocol, finalizing its backup at `Back(V)`. If `Back(F)` has a temporary backup from `F`, it kills the program. Otherwise it restores the new backup, which includes the stolen tasks.
- Failure of `V`: Depending on time, `Back(V)` restores the old state, restores the new state, or kills the program.

Killing the program when `tmpBackup` ≠ `null` can be avoided by some additional handshaking between `Back(V)` and `F`, or between `Back(F)` and `V`, in the event of a failure. This is done in our asynchronous fault tolerance scheme, which is described in Sect. 5. In the synchronous scheme, we stayed with the simpler protocol.

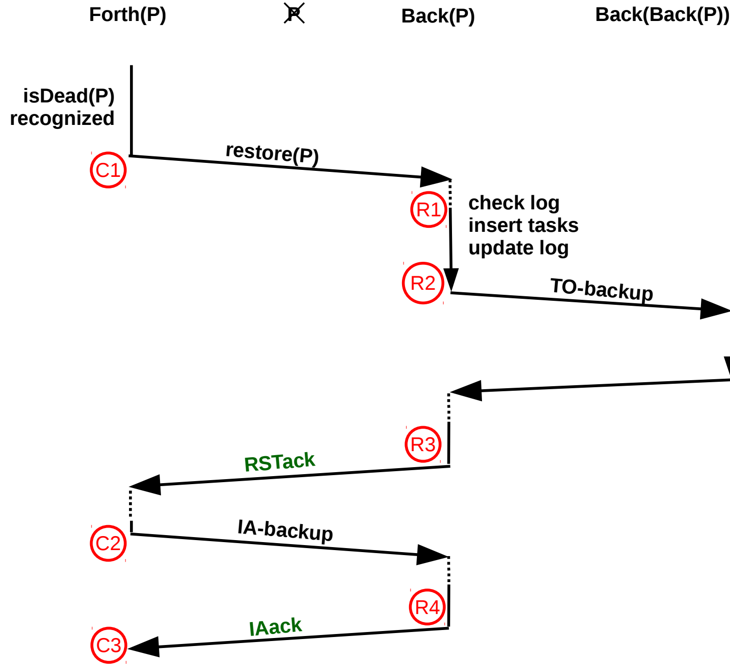

Fig. 4.2. *Restore Protocol of the Synchronous Scheme*

Depending on context (e.g. overall running time), the lower robustness of this scheme may be sufficient since typical steal rates are low [17], and place failures are rare, such that their coincidence is unlikely.

Remark on Implementation. In a few cases, the synchronous scheme uses asynchronous messages, similar to the original GLB. The original GLB marks these messages with `@Uncounted`, such that `finish` ignores these activities in its bookkeeping to save time. Unfortunately, `@Uncounted async` discards exceptions [18]. Since we need exceptions for fault tolerance, we had to eliminate some of the `@Uncounted` annotations.

5. Asynchronous Fault Tolerance Scheme. As noted in Sect. 3, each worker has a three-phase structure, which we have implemented by the following loop:

```
while (nTasks > 0 || stealFailed) {
  processUpToNTasks();
  Runtime.probe();
  processRecorded();
}
```

Here, `stealFailed` is true iff all $w + z$ steal attempts failed. At the call to `Runtime.probe()`, all pending remote activities are run. They may record requests, e.g. steal requests, in the worker's data structures. Recorded requests are carried out by the worker in `processRecorded`.

5.1. Steal Protocol. The asynchronous steal protocol is depicted in Fig. 5.1. A victim `V` may have more than one thief, but for clarity, only one thief `F` is depicted here. As compared to the protocol in Sect. 4, this protocol has been designed to strictly minimize communication. This manifests in minimalist handshaking and handling multiple steal requests together. These aspects will be elaborated later.

First, let us look at a positive side effect of stealing: redundancy. When tasks are sent from `V` to `F`, they are copied, and thus exist twice. Inspired by resilient divide-and-conquer algorithms [9, 10], we exploit the redundancy for fault tolerance. Thus, the backups are not updated to account for task movement, or at least not immediately. Instead, the stolen tasks are kept at `V`, and only a link to `V` is sent to `Back(F)`. If `F` dies, `Back(F)` obtains the tasks from `V`. This way, the backup volume is reduced.

Nevertheless, excessive scattering of a worker's data must be avoided, since it would increase the probability

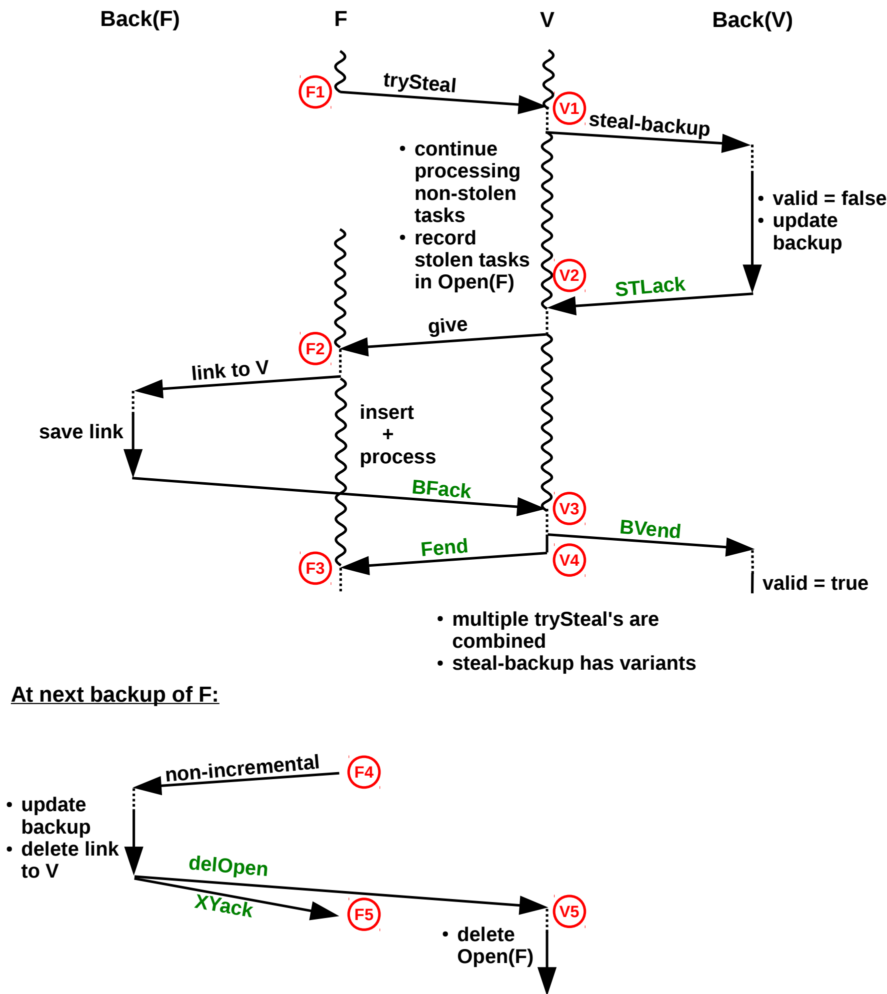

Fig. 5.1. *Steal Protocol of the Asynchronous Scheme*

of coincident failures. Therefore, as shown at the bottom of Fig. 5.1, the links are replaced by real tasks at any next backup. The backup, which may be scheduled at `F` for whatever reason, incorporates the tasks behind the links (if not finished yet). It is called an *afterMerge*-backup. After this backup, a `delOpen` message is sent to `V`, to release the tasks.

Because of the algorithm's three-phase structure, it is well possible that multiple steal requests are received at `V` at the same time. Thus, in Fig. 5.1, a victim `V` may have more than one thief, but for clarity only one of them is depicted. To reduce the number of messages, multiple steal requests are handled together in a so-called *transaction*. Transactions are numbered by *tan*s, which are assigned consecutively by each worker i. Consequently, a pair (i, tan) is system-wide unique.

First, `V` runs through the thieves in some particular order, which is defined later. For each thief `F`, it determines the tasks to be sent. According to the lifeline scheme, the first thief gets about half of the tasks, the second a quarter, and so on. Any remaining requests are rejected. The transaction is formed from those thieves that get tasks. Their identities are kept in a list at `V` during the transaction.

The tasks destined for the different thieves are removed from the local pool, and inserted into a local data structure at `V`, called `Open`. It is a P-size array of lists, where each entry `Open(F)` holds the tasks to be sent from `V` to `F`. Additionally, references to the task groups of the current transaction are saved in another list, to

speed up access.

Before giving the tasks to thieves, `V` informs its backup partner. In Fig. 5.1, this is called `steal-backup`. Note that a single backup is sufficient for all steal requests of a transaction. A steal-backup differs from other types of backup in that the tan and a list of thieves are included.

Note that the stolen tasks are omitted from the steal-backup, which reflects our approach of relying on the redundancy of stealing. Since at least half of the tasks are stolen, the backup volume is significantly reduced this way.

A steal-backup may be an *afterMerge* backup. After the steal-backup, `V` and its backup are synchronized. Thus, the counter for the kn time period may be reset, which relativizes the steal backup's expense.

During the steal-backup, `V` goes on processing tasks, to avoid loosing time while waiting. When the backup is finished, as signalled by the receipt of the `STLack` message, `V` gives the task groups to the respective thieves. This operation runs fast, since the task groups have already been recorded before. Then, `V` returns to work, again, to not loose time waiting.

At this point, `V` can safely give away the tasks, since `Back(V)` is able to and will take care of consistency. Briefly stated, if `V` dies, `Back(V)` will contact all thieves of the current transaction (whose identities have been stored), and make sure they have received their tasks. Usually they have, or will after some short waiting (except if there is a second failure). Nevertheless, the time between receiving the steal backup and the `BVend` message is somewhat critical for `Back(V)`, since it does not hold `V`'s state completely. Therefore, we denote this time period as *queasy* and mark it by `valid=false`.

Note the asymmetry between the victim and thief sides: `F` is involved in a single give of the transaction, whereas `V` may give tasks to multiple thieves. As stated before, `F` sends the steal requests consecutively, but can nevertheless receive multiple `give` messages at about the same time. They are handled in any order. It is not necessary to wait for the final `Fend` message of the first `give` before processing the next. Thus, there may be multiple outstanding `Fend` messages. `F` keeps them in a list, called `OpenFend`. Only when this list is empty, a backup may be sent from `F` to `Back(F)`. The resulting delay is limited, since `F` may receive tasks from at most z workers.

Bookkeeping of all participants makes use of the tans. They are included in all messages of the steal protocol, and saved wherever appropriate. For instance, at `V`, they are added to each task group. Unlike the other participants, `F` saves the tans of merged task groups permanently. It needs the information since, in case `V` dies after (V4), it must be able to answer `Back(V)`'s inquiry. Fortunately, there is an efficient scheme for permanent storage: Since `F`'s at most $w + z$ steal requests go to different victims, subsequent transactions with the same victim are processed in the order of their tans. Therefore, it is sufficient to keep each `V`'s most recent received tan.

For each `give`, `F` first sends the respective link to `Back(F)`, and then inserts the tasks. Right afterwards, `F` starts processing the tasks. There is no problem about returning to work since, if `F` dies before (F4), `Back(F)` resets to a previous state anyway, and if `F` reaches (F4), the results of work are correctly reflected in the new backup.

As noted above, the handshaking in Fig. 5.1 has been designed to be minimalistic. In particular, `Back(F)` only reports to `V`, but not to `F`. When `V` receives a `BFack` message, it immediately sends `Fend` to the respective thief. Moreover, it removes `F` from the thieves-list of this transaction. It easily recognizes when all `F`'s have sent their `BFack`. Then, the transaction is over, and `V` signals that to `Back(V)` via a single `BVend` message.

5.2. Restore Protocol. Restore presupposes timely failure notification, which has been explained in Sect. 3. In the following, we assume that `Forth(P)` has recognized P's failure. The restore protocol is depicted in Fig. 5.2. Note that `Forth(P)`, P, etc. need not have successive numbers, but workers in-between may have been restored.

The protocol is time-critical between (C1) and (R4), since `Forth(P)`'s data are unsecured. Therefore, `Forth(P)` and `Back(P)` run the protocol in so-called *emergency mode*. In this mode, processing of tasks pauses, for higher reactivity. Moreover, only urgent and short messages are handled. The other messages must be received, but they are only stored and handled later. The loss in efficiency due to emergency mode can be neglected, since we assume that failures are rare.

As shown in Fig. 5.2, the protocol resembles the restore protocol of the synchronous fault tolerance scheme,

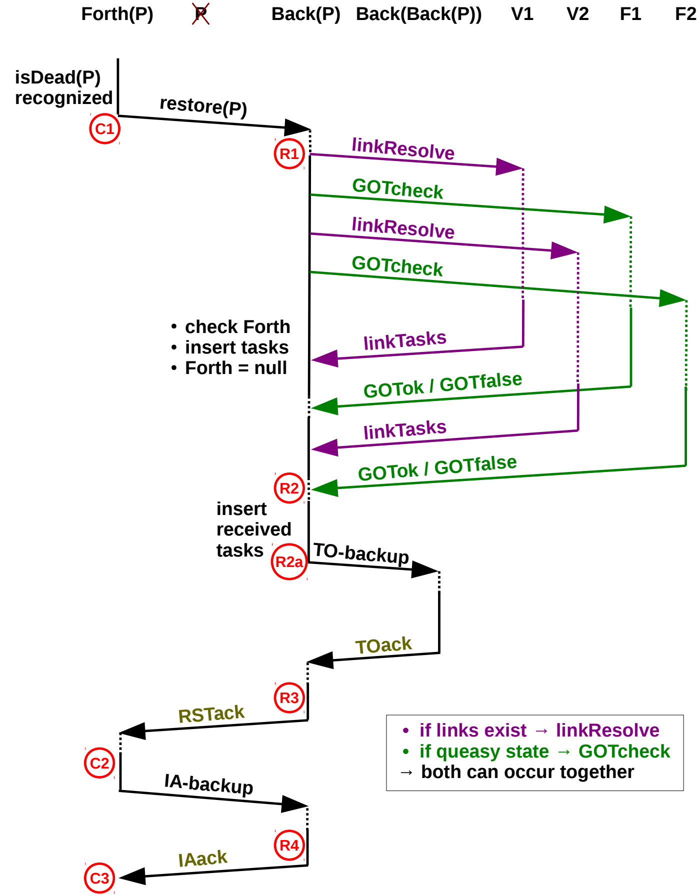

FIG. 5.2. *Restore Protocol of the Asynchronous Scheme*

but accounts for the fact that tasks referenced by links must be collected. Also, to improve robustness, failure of `V` in queasy state does not automatically lead to program abort. Instead, `Back(V)` checks whether the tasks have been taken over by the thief side. Only if the tasks are lost, the program is killed.

The protocol begins with a restore message from `Forth(P)` to `Back(P)`. Upon receipt, as explained at the end of Sect. 3, `Back(P)` makes sure that `P = Forth`, and that the source equals `Forth(Forth(P))`. Since failures may happen at any time, three particular situations must be taken care of:

1. P fails between (V1) and (V4) in the steal protocol, i.e., `Back(P)` is in queasy state.
2. P fails between (F2) and (F5), and `Back(P)` has saved links.
3. P fails when `Forth(P)` and/or `Back(P)` are already involved in another restore protocol.

Situations 1 and 2 may appear together, since a worker is allowed to receive tasks between (V2) and (V3). In situation 1, `Back(P)` contacts all thieves of the current transaction, called `GOTcheck` in Fig. 5.2. They check whether they have received tasks for the corresponding tan, otherwise wait for a moment in case the message is late, and then respond (called `GOTok/GOTfalse` in the figure). The restore can only be performed if all answers are `GOTok`. Otherwise, `Back(P)` aborts the program.

In situation 2, `Back(P)` sends a `linkResolve` message to each former victim. Usually, the victim finds the respective tasks in `Open(P)` and sends them back in a `linkTasks` message. Occasionally, though, the victim has recognized P's failure before `Forth(P)` did (from a timeout on an outstanding `BFack`). In this case, the victim has already taken back the tasks, i.e., it re-inserted them into its own pool and removed them from `Open(P)`. Consequently, the tasks are not available anymore. Interestingly, such orphaned links are no problem: The victim just sends back `null` instead of a task group, and therewith informs `Back(P)` that it has taken care of

the tasks. One may ask why the victim takes back the tasks when observing timeout. The reason is that `F` may have failed before (F2), and, thus, there is no link.

Situation 3 stands for several sub-cases, which will be discussed below. Now, let us look at a normal protocol run. After having sent the `linkResolve` and `GOTcheck` messages, `Back(P)` waits for answers. During that time, in fact, it performs the `Forth` checks discussed earlier. Moreover, it inserts the tasks from the saved backup into its own pool, as it does with the tasks arriving via `linkResolve` in course of time.

Sending out the messages before deciding whether to perform the restore at all may appear counter-intuitive. This order is more economical, though, since it avoids loosing the waiting time, and time-critical messages are sent earlier this way. Changing the order does not compromise correctness: If `Back(P)` remains alive, the order does not matter. If `Back(P)` dies before performing the `Forth` checks, two successive workers are gone, and `Back(Back(P))` kills the program anyway. While details are omitted for brevity, considerations like that determine the ordering of actions throughout our algorithm.

When all outstanding `linkTasks` and `GOTok` messages have been received and the tasks have been inserted (denoted (R2a) in the figure), `Back(P)` has successfully restored P. This corresponds to (R2) in Fig. 4.2. Like there, the protocol finishes with `TO-` and `IA-backups`.

The most important sub-cases for situation 3 are handled as follows:

- `Back(P)` dies during the protocol:After recognizing the failure,`Forth(P)` figures out that `Back(Back(P))` is the next place alive along the ring, and sends a restore message. If `Back(Back(P))` has already received `Back(P)`'s `TO-backup`, it performs the restore, otherwise it aborts the program.
- `Forth(P)` receives a `restore` between (C1) and (C2): `Forth(P)` kills the program.
- `Back(P)` recognizes failure of `Back(Back(P))`: `Back(P)` kills the program.

In the second and third cases, the program could in principle be continued by nesting protocol execution. This would, however, complicate the program. Since we have assumed unfavourably correlated failures to be unlikely, we instead abort the program.

5.3. Communication Structure. As noted before, the actor scheme allows us to handle incoming messages in a predefined order. Moreover, messages can be easily prioritized or delayed until the next iteration of the main loop. The order is accurately defined for all message types of our algorithm but, for brevity, the following list is restricted to a few typical representatives:

- Handshaking and link management messages are handled immediately (e.g. `noTasks`, `BFack`, receipt of a link, `linkResolve` and `GOTcheck`)
- Arrival of a `TO-backup` or `IA-backup` has the highest priority of recorded requests.
- Arrival of `restore` comes thereafter, since a `TO-backup` may increase the chances for successful restore.
- Upon arrival of `give`, the link is sent immediately, but the (more time-consuming) merge action is recorded.
- Steal requests come after merge, so that the steal backup replaces the links, and the received tasks can be distributed.
- Regular backups have lowest priority.

6. Experiments. Experiments were conducted on an Infiniband-connected cluster, where each node comprises two 8-core Intel Xeon E5-2760 CPUs as well as 32 GB main memory. All experiments used one place per node on up to 128 nodes. We deployed X10 version 2.5.2 (SVN, revision 29421) and GCC 4.8.4 to compile the programs. Resiliency was only switched on for the fault-tolerant program versions.

We used two benchmarks: Unbalanced Tree Search (UTS) and Betweenness Centrality (BC). Both are included as samples in the X10/GLB distribution. UTS is a well-known benchmark that counts the number of nodes in a highly irregular tree, which is constructed on the fly from node descriptors [12]. Each descriptor encodes a subtree and naturally corresponds to a task. We used geometric distribution for the tree shape. BC considers the set of all shortest paths in a graph, and for each node computes the number of shortest paths running through it [13]. GLB uses UTS as an example for dynamic task pool initialization, and BC for static initialization.

For the fault-tolerant frameworks, the sample codes of UTS and GLB were slightly adapted. In particular, the BC example was extended by shadows, as explained in Sect. 3.

Table 6.1
Experimental results for the Parameter n and k

Benchmark	Configuration	Framework	determined n and k
UTS	small	GLB	$n = 512$
		FTGLB	$n = 4\,096$, $k = 65\,536$
		FTGLB-Actor	$n = 8\,192$, $k = 65\,536$
	large	GLB	$n = 16\,384$
		FTGLB	$n = 32\,768$, $k = 65\,536$
		FTGLB-Actor	$n = 65\,536$, $k = 65\,536$
BC	small	GLB	$n = 512$
		FTGLB	$n = 65\,536$, $k = 512$
		FTGLB-Actor	$n = 32\,768$, $k = 1\,024$
	large	GLB	$n = 16\,384$
		FTGLB	$n = 65\,536$, $k = 512$
		FTGLB-Actor	$n = 65\,536$, $k = 512$

Both UTS and BC were run with a small and a large configuration, to account for different computation-to-communication ratios. In the following, b denotes the branching factor, d the tree depth, s a random seed, and N the number of graph nodes:

- small UTS: $d = 13$, $b = 4$, $s = 19$
- large UTS: $d = 17$, $b = 4$, $s = 19$
- small BC: $N = 2^{14}$, $s = 2$
- large BC: $N = 2^{16}$, $s = 2$

We compared three program versions:

- the original GLB code from the X10 distribution (GLB),
- our synchronous GLB version from Sect. 4 (FT-GLB), and
- our asynchronous GLB version from Sect. 5 (FTGLB-Actor).

Experiments were grouped into two stages. In the first stage, we determined the optimal n and k values for the `process(n)` calls and the kn backup intervals. In the second stage, we measured performance with the best n and k values obtained before. All experiments were repeated three times, and the average was taken.

In the first stage, we started the small UTS and BC configurations on 8 nodes, and the large configurations on 32 nodes, varying n. These place numbers have been observed before to lead to good performance. Although [1] reports on scalability to many more places, we were not able to achieve such speedups with our configurations and hardware.

The best n and k values are depicted in Table 6.1. The fault-tolerant program versions prefer higher n, due to their increased overhead. Although steal rates are similar for all programs, these versions need to perform additional actions to, e.g., write backups. The k values are not important for UTS, since steals, and therefore steal backups, are frequent. Therefore, any sufficiently large k value performs well. For BC, we chose k such that a regular backup is written roughly every 10 seconds.

Results of stage 2 are depicted in Figs. 6.1 and 6.2 for the small configurations, and in Figs. 6.3 and 6.4 for the large configurations.

For a deeper analysis, we divided a typical run into three phases:

- Setup and initial work distribution: Each place writes its initial backup. Dynamic work distribution requires stealing.
- Steady state: The nodes work mostly independently, the steal rate is low.
- Final stage: More and more places run out of work and steal from each other, increasing communication.

For the small configurations, the frameworks spent a high portion of time in the final stage, which led to a high communication-to-computation ratio. The huge overhead of up to 655% for FTGLB as compared to GLB in Figs. 6.1, 6.2 and 6.4 indicates that the synchronous scheme has deficiencies. They have been largely resolved with the FTGLB-Actor framework, which exhibits almost the same performance as GLB in the UTS runs. In

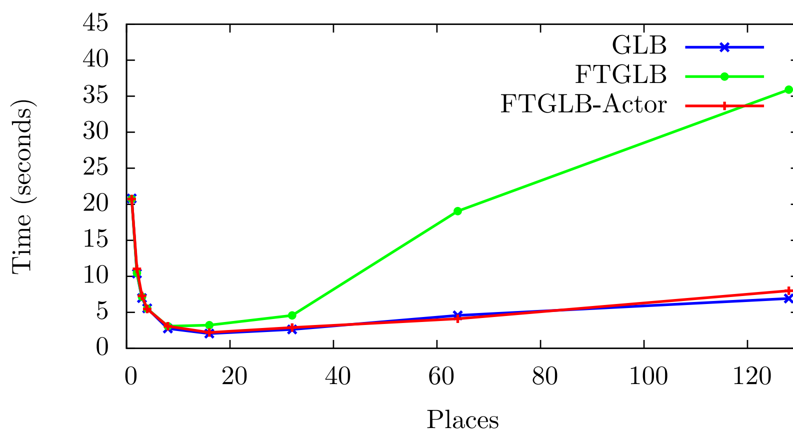

Fig. 6.1. *Experimental results for the small UTS configuration.*

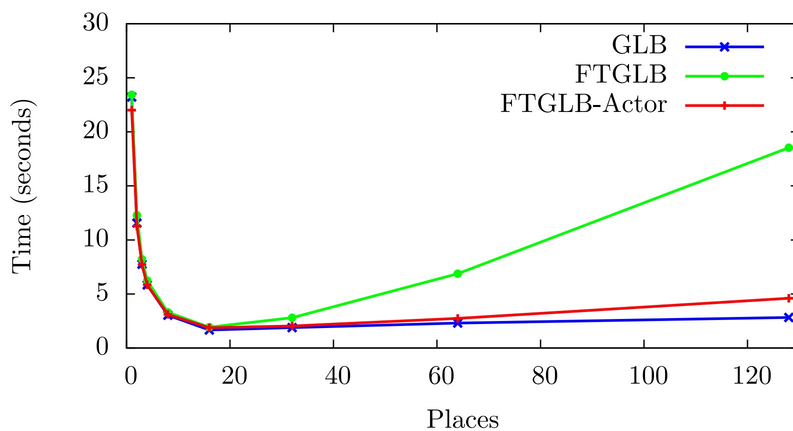

Fig. 6.2. *Experimental results for the small BC configuration.*

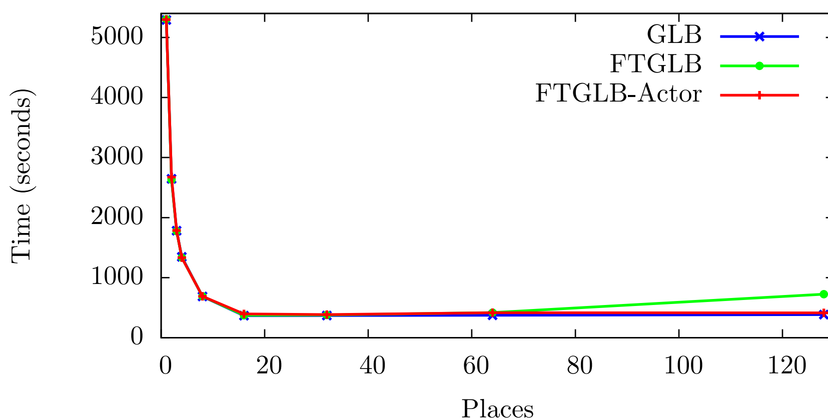

Fig. 6.3. *Experimental results for the large UTS configuration.*

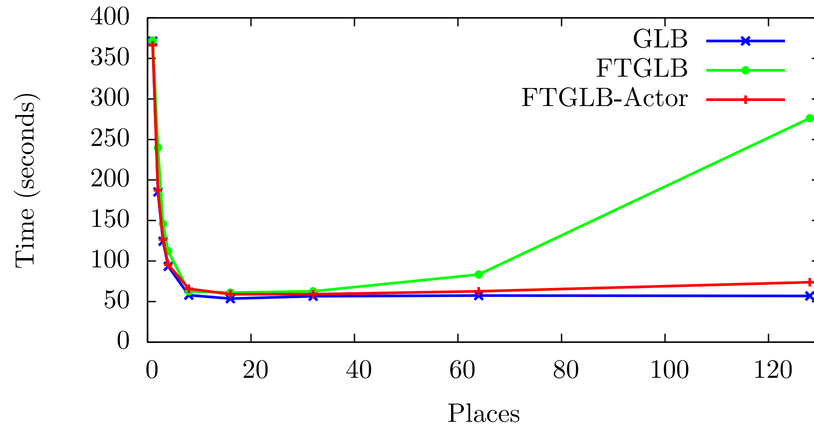

Fig. 6.4. *Experimental results for the large BC configuration.*

the BC runs, there is a notable difference of up to 62.5% between GLB and FTGLB-Actor in Figs 6.2 and 6.4. It can be explained by the need to manage shadows.

To measure the overhead of our restore protocols, we started three runs on 5 and 9 nodes for each configuration and benchmark, crashed one place shortly after the start of the task execution, and compared the execution time with that on 4 and 8 places, respectively. The overhead was about 6% in all cases.

7. Related Work. As already stated in the introduction, related work differs from ours in the task model used. First, MapReduce systems such as Hadoop [8] assume the set of tasks to be fixed. Ciel [19] extends MapReduce by permitting tasks to generate new tasks, but still task management is centralized at a master process. Second, divide-and-conquer algorithms exploit references between stolen children and their parents for fault tolerance [9, 10], whereas we discard parents and maintain only one partial result per worker. Third, side effects in tasks can be tackled by tracking operations [20].

Outside fault tolerance, load balancing and task pools are a long-standing area of intensive research, e.g. [1, 6, 14, 21]. In [22], an X10 keyword for mobile activities is suggested that brings activities and global load balancing together.

Resilient X10 has been deployed in other applications [4]. For instance, a fault-tolerant `DistArray` data structure [23] bases recovery on regular backups, similar as we do. Less closely related to our work, transient faults in task pools can be detected and handled by replication and voting [24].

8. Conclusions. This paper has introduced two different task pool algorithms that can tolerate any number of permanent place failures, except failure of the first place. The algorithms are based on backups of the local task pool contents, which are regularly written to the main memory of a neighboured place. Important aspects have been failure notification, as well as steal and restore protocols. Especially the protocols differ significantly between the algorithms. Most importantly, the first algorithm uses synchronous, and the second asynchronous communication. To reduce complexity, the second algorithm deploys an actor-like communication structure. Moreover, it exploits stealing-inherent redundancy, and requires less handshaking. Both algorithms are conservative in that a computed result is guaranteed to be correct, at the price of halting the program in a few rare cases. There are more such cases for the first than for the second algorithm.

The algorithms have been implemented in the GLB framework of X10. In experiments with UTS and BC, we observed significant overheads for the first algorithm, but the second algorithm's performance was close to that of non-fault-tolerant GLB.

There are several directions for future research. First, we will finish the implementation of the third scheme, which was mentioned in Sect. 1. Beyond that, it would be interesting to apply our approach to other task pool algorithms, especially to algorithms that allow multiple workers per place and deploy concurrent data structures. Also, it would be interesting to implement the fault tolerance schemes in other programming systems, e.g. in

MPI with user level failure mitigation [26].

REFERENCES

[1] V. Saraswat, P. Kambadur, S. Kodali *et al.*, "Lifeline-based global load balancing," in *Proc. ACM Symp. on Principles and Practice of Parallel Programming*, 2011, pp. 201–212.
[2] M. Bungart, C. Fohry, and J. Posner, "Fault-tolerant global load balancing in X10," in *Proc. SYNASC Workshop on HPC Research Services*, 2014, pp. 471–478.
[3] *X10 Homepage*. [Online]. Available: x10-lang.org
[4] D. Cunningham, D. Grove, B. Herta *et al.*, "Resilient X10: Efficient failure-aware programming," in *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2014, pp. 67–80.
[5] W. Bland, P. Du, A. Bouteiller *et al.*, "A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI," in *Proc. Euro-Par*. Springer LNCS 7484, 2012, pp. 477–488.
[6] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," *ACM SIGPLAN Notices (PLDI)*, vol. 33, no. 5, pp. 212 – 223, 1998.
[7] *Chapel Homepage*. [Online]. Available: chapel.cray.com/
[8] *Hadoop Homepage*. [Online]. Available: http://hadoop.apache.org/
[9] R. D. Blumofe and P. A. Lisiecki, "Adaptive and reliable parallel computing on networks of workstations," in *Proc. USENIX Annual Technical Symp.*, 1997.
[10] G. Wrzesinska, R. V. V. Nieuwpoort, J. Maassen, and H. E. Bal, "Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid," in *Proc. Int. Parallel and Distributed Processing Symp.*, 2005.
[11] G. Wrzesiska, A.-M. Oprescu, T. Kielmann *et al.*, "Persistent fault-tolerance for divide-and-conquer applications on the grid," in *Proc. Euro-Par*. Springer LNCS 4641, 2007, pp. 425–436.
[12] S. Olivier, J. Huan, J. Liu *et al.*, "UTS: An Unbalanced Tree Search benchmark," in *Proc. Workshop on Languages and Compilers for High-Performance Computing*. Springer LNCS 4382, 2006, pp. 235–250.
[13] *HPCS Scalable Synthetic Compact Applications #2: Graph Analysis*. [Online]. Available: http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.0.pdf
[14] U. A. Acar, A. Chargueraud, and M. Rainey, "Scheduling parallel programs by work stealing with private deques," *ACM SIGPLAN Notices (PPoPP)*, vol. 48, no. 8, pp. 219–228, 2013.
[15] C. Fohry, M. Bungart, and J. Posner, "Towards an efficient fault-tolerance scheme for glb," 2015, to appear.
[16] G. A. Agha and W. Kim, "Actors: A unifying model for parallel and distributed computing," *Journal of Systems Architecture*, vol. 45, no. 15, pp. 1263–1277, Sep. 1999.
[17] V. Kumar, D. Frampton, S. M. Blackburn *et al.*, "Work-stealing without the baggage," *ACM SIGPLAN Notices (OOPSLA)*, vol. 47, no. 10, pp. 297–314, 2012.
[18] *X10 Distribution*. [Online]. Available: http://sourceforge.net/p/x10/mailman/message/24998422/
[19] D. G. Murray, M. Schwarzkopf, C. Smowton *et al.*, "CIEL: a universal execution engine for distributed data-flow computing," in *Proc. USENIX Conf. on Networked Systems Design and Implementation*, 2011.
[20] W. Ma and S. Krishnamoorthy, "Data-driven fault tolerance for work stealing computations," in *Proc. ACM Int. Conf. on Supercomputing*, 2012, pp. 79–90.
[21] J. Dinan, D. B. Larkins, S. Krishnamoorthy *et al.*, "Scalable work stealing," in *Proc. SC Conf. on High Performance Computing, Networking, Storage and Analysis*, 2009.
[22] J. Paudel, O. Tardieu, and J. N. Amaral, "On the merits of distributed work-stealing on selective locality-aware tasks," in *Proc. Int. Conf. on Parallel Processing*, 2013, pp. 100–109.
[23] K. Kawachiya, "Writing fault-tolerant applications using resilient X10," IBM Research Tokyo, Tech. Rep. RT0960, Apr. 2014.
[24] Y. Wang, W. Ji, F. Shi, and Q. Zuo, "A work-stealing scheduling framework supporting fault tolerance," in *Proc. Design, Automation and Test in Europe*. EDA Consortium / ACM DL, 2013.
[25] *OpenMP Homepage*. [Online]. Available: http://openmp.org
[26] *Process Fault Tolerance (Unofficial Draft for MPI-Standard)*, 2014. [Online]. Available: https://svn.mpi-forum.org/trac/mpi-forum-web/raw-attachment/ticket/323/ticket323.pdf

Edited by: Dana Petcu
Received: May 20, 2015
Accepted: Jun 24, 2015