Fault Tolerance for PCJ

Michał Szynkiewicz

Faculty of Mathematics and Computer Science Nicolaus Copernicus University, Toruń, Poland

Abstract. Nowadays, distributed computations are often run on large amounts of nodes for a significant amount of time. Having a 24 hour long computation on 1000 nodes means almost 3 years of computations. As calculated in [1], with 6 month mean time between failures, it gives less than 1% chance of success.
PCJ library is a high-performance parallel computing library for Java implementing PGAS model.
This paper describes changes introduced in PCJ to provide basic fault tolerance and fault tolerance strategies that are planned to be implemented in future.

1 Introduction

Before the changes described in this paper, any network or hardware failure resulted in hanging of a PCJ-based program.

The changes introduce *Ignore failure* fault tolerance policy. This is a minimum-overhead, no-checkpoint strategy.

For programs that do not require results from all nodes to finish, like Monte Carlo randomized algorithms, the strategy is enough to finish calculations properly. For other programs, the strategy provides a useful base on which the programmer may build a dedicated solution (e.g. replaying failed nodes work on other node).

2 PCJ library

PCJ library is a Java implementation of PGAS model. It is fast, lightweight and it ships as a single jar.

The smallest unit of computation in PCJ is called a *thread*. Each *thread* has its own local address space and shared address space. The owner *thread* accesses local and global address spaces variables as usual Java fields.

Multiple *threads* may be run on one physical machine, within a single Java Virtual Machine (JVM). A *thread* or group of *threads* run on a single JVM are called a *node*.

PCJ distinguishes one node, namely *node 0*. This is the start node for all computations.

Following is a description of PCJ features. For more details please see [3].

2.1 Synchronization

Threads synchronization in PCJ is realized by `barrier`. Barrier can be made either across all threads or two threads. In first case each task has to call `barrier` method. Barrier is managed by *node 0* and realized in two steps. First, each node on which all threads have called `barrier` informs *node 0* that it has reached `barrier` and starts waiting. When *node 0* gathers barrier confirmations from all nodes, it releases the barrier by broadcasting a specific message to all nodes. Broadcasting is realized in an efficient way described later.

2.2 Shared memory

Fields that should be put into shared address space have to be annotated in `@Shared` annotation. Aformentioned fields are accessible from other threads via following methods:

- `get(threadId, variableName)`, `getFutureObject(threadId, variableName)` - read the value of given variable in a synchronous and asynchronous fashion
- `put(threadId, variableName, value)` - asynchronously update variable value in given thread.
- `broadcast(variableName, value)` - asynchronously update variable value in all threadsWhat makes `broadcast` fast is the fact that it does not update the variable value one by one,. What makes `broadcast` fast is the fact that it does not update the variable value one by one. Underlying communication is organized in a balanced binary tree which means that the update will reach each node in at most *O(log n)*, where n is number of nodes. This is also the way that barrier finish message is broadcasted.

`waitFor(variableName)` and `monitor(variableName)` can be used to lock the current thread until other thread updates given variable on this node.

3 Related work

As argued in [4], failures are becoming a norm in todays supercomputing systems. Hence, recently more and more parallel computing libraries start to provide fault tolerance features.

Authors of [4] propose a checkpoint/restart fault tolerance policy for OpenSHMEM - PGAS library with C and FORTRAN APIs. In the suggested model `shmem_checkpoint_all()` method is introduced. Call to `shmem_checkpoint_all()` will result in copying all of its shared (symmetric) data to another node. The method needs to be called on all nodes. Then, in case of a failure, the failed nodes can be restarted and job of all nodes can be started from the latest checkpoint. The power of this solution comes with the cost of synchronization of all nodes on every checkpoint.

[1] describes a fault tolerance policy for X10 language, asynchronous PGAS (APGAS) implementation. Language after changes is called Resilient X10. Resilient X10 introduces resilient storage that allows access to nodes data even

if the node has failed and enables programmer to handle failures by throwing DeadPlaceException when node fails.

TODO: maybe a few words about things from "Fault Tolerance for Remote Memory Access Programming Models"

4 Fault tolerance

Described solution assumes that *node 0* never dies. Although this makes the solution not entirely fault tolerant, it is a common practice to do it. *node 0* is the place where typically results are gathered. Hence if the computation would continue after its failure, it would require additional work to find where the results were stored. Moreover the probability of *node 0* failure is much smaller than the probability of failure of one of all nodes.

4.1 API

The main concern for the API changes provided by the implementation is not to break currently implemented programs.

Most of the changes are transparent for the programmer.

In order to let programmer check which threads failed, `getFailedNodes()` method have been introduced.

The implementation introduces `NodeFailedException`, which is thrown by methods which cannot finish properly because of node failure. The fact that it is a runtime exception makes programs written in non-fault-tolerant versions of PCJ runnable also on this version.

Methods that might throw `NodeFailedException` are:

- `get/getFutureObject` - when it tries to access data from a failed node. In case of `get` the exception is thrown right away. In case of `getFutureObject` the exception might be thrown right away or when program tries to read the result of future object.
- `waitFor` - on any node failure. `waitFor` is used to wait for an update of a variable. PCJ does not have any information which node should update the variable. In order to let the program work - the exception is thrown. The programmer may then check via `getFailedNodes()` if the node that should update the variable is still alive and invoke `waitFor` again.
- `put` - on attempt to update a variable on a node that failed

4.2 Implementation

The main elements of implementation are: monitoring nodes for failure, adjusting configuration after node failure and replaing the messages that might have been lost.

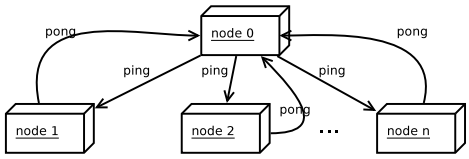

Fig. 1. Node monitoring

Node monitoring The immortal *node 0* is monitoring other nodes by sending PING message and waiting for PONG. If, for some node, sending PING fails, or the node does not answer with PONG for a long (configurable) time, it is assumed failed. Please see Fig. 1 Node monitoring.

Reconfiguration Let's denote the failed node *node k*. When *node 0* discovers that *node k* failed it does the following things:

1. removes the failed node from configuration - not to try to send any data to it
2. if there is a barrier in progress - removes the node from the barrier. If it is the last node the barrier is waiting for - the barrier is finished and its finish is propagated to other nodes. See Fig. 2 Continuing barrier after node failure.
3. the communication tree is updated. If, in the communication tree, *node k* has two children: *node i* and *node j*, they are reattached to another node that is still active. This is realized in a following manner: first, *node 0* finds a leaf in a communication tree. Then *node i* and *node j* are attached to the leaf as its children. Please see Fig. 3 Fixing communication tree Note that after this operation the tree is no longer fully balanced.

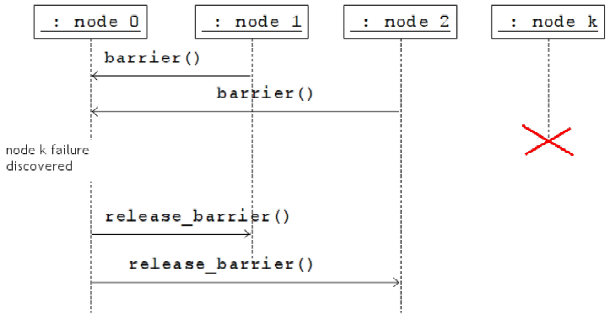

Fig. 2. Continuing barrier after node failure

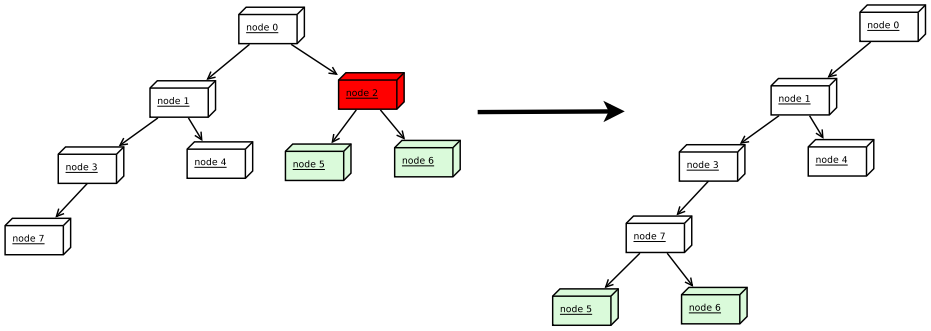

Fig. 3. Fixing communication tree

Replaying communication Node failure can cause loss of underlying communication.

Let's take a look at `barrier`. As described in 2.1, when barrier is reached by all threads, *node 0* broadcasts a message that releases the barrier to all nodes. When *node k* fails during broadcast and it has *node i* and *node j* as children, *node i* and *node j* will not get this message and hang.

To prevent this situation *node l*, which is established a new communication parent for *node i* and *node j*, has to replay communication. Since a node can never know in advance when it will become a communication parent, every node stores all broadcast communication that it gets to be able to resend. Messages are stored for some amount of time and evicted when the time passes.

When *node i* and *node j* are attached to *node l*, *node l* sends all broadcast messages to *node i* and *node j*. It might happen that a message was already processed on e.g. *node i*. Each message is given a unique identifier. Based on this identifier, PCJ internals figure out if the message has been already processed or not. If it has not, the message is processed and sent further in the communication tree. If it has, the message is only passed through.

4.3 Performance overhead

To check performance overhead of introduced changes, a test that invoked barriers one by one was executed. The test was performed on a 4 core machine with 20 PCJ nodes.

Fig. 4 Performance comparison shows results of the test. Each result is a median of three test runs.

TODO: analyze

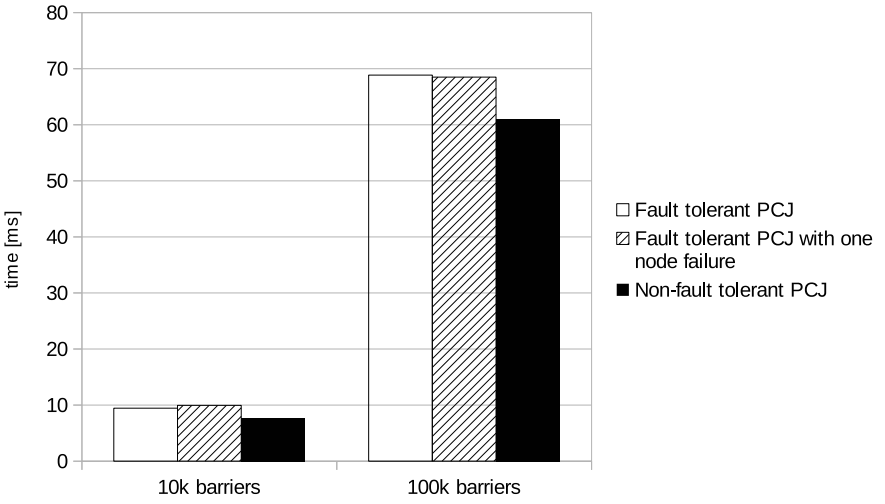

Fig. 4. Performance comparison

5 Future work

5.1 Resilient storage implementation

6 Conclusion

References

1. Leslie Lamport, Dave Cunningham, Dave Grove, Ben Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, Olivier Tardieu *Resilient X10. Efficient failure-aware programming*, Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming
2. Marek Nowicki, Piotr Bała *PCJ - New Approach for Parallel Computations in Java* Applied Parallel and Scientific Computing, 11th International Conference, PARA 2012, Helsinki, Finland, June 10-13, 2012, Revised Selected Papers
3. Marek Nowicki, Piotr Bała *PCJ. Parallel Computing in Java*, 2013, [Online]. Available: `http://pcj.icm.edu.pl/c/document_library/get_file?uuid=e8b5a416-644d-43d0-a19c-74e14555ef96&groupId=43801` [2015, April 14]
4. Pengfei Hao, Pavel Shamis, Manjunath Gorentla Venkata, Swaroop Pophale, Aaron Welch, Stephen Poole, Barbara Chapman, *Fault Tolerance for OpenSHMEM*, PGAS/OUG14 Available: `http://nic.uoregon.edu/pgas14/oug_submissions/oug2014_submission_12.pdf` [2015, April 17.]