Correcting Response Failure Errors in Multi-Objective Optimisation in Unreliable Distributed Computing Environments

Timothy Rawlins and Andrew Lewis, Affiliate Member, IEEE

Abstract—Population-based, multi-objective optimisation algorithms are increasingly making use of distributed, parallel computing environments. In these cases it is a commonsense precaution to consider the possibility of a variety of failures. In particular, errors caused by response failures are more prone to arise than in homogeneous parallel computers. While masking errors using redundant computation is simple and reasonably reliable, it is expensive in terms of the computing resources required. An alternative approach is presented that uses a Byzantine agreement methodology, utilising only results already computed. In computational experiments it has demonstrated ability to correct errors, and salvage useable results from unreliability, distributed computing environments. With increasing reliance on computing resources provided and operated by external agencies, error detection and correction can be expected to become more important to a range of applications.

Index Terms—Fault tolerance, distributed computing, multi-objective optimisation.

I. INTRODUCTION

The use of simulation and computational modelling in the engineering design process has become increasingly commonplace as the power of computers available to designers has increased and their cost significantly decreased. A growing trend is the availability of parallel computer architectures, whether in a single machine e.g. multi-core, or through the use of computational grids or cloud computing to access resources operated and made available by external agencies. Using these increasingly powerful resources, computational design optimisation, governed by specified design objectives, has become a realisable alternative to trial and error evolution of improved designs.

Multi-Objective Particle Swarm Optimisation (MOPSO) [1], [2], is an algorithm that uses a population of trial solutions to find a set of (near) optimal solutions for problems that have two or more, possibly competing, objectives. Instead of returning a single solution to the design problem, a set of Pareto-optimal solutions are generated, representing the trade-offs between the objectives, a complex task that introduces further difficulties in determining what “optimal” designs are, and how to obtain them [3], [4].

Because MOPSO problems usually involve searching through large, multi-dimensional parameter spaces, and each point in the parameter space can be evaluated individually, parallel computing is a natural match to the computational needs of the optimisation process, and a variety of approaches have been explored [3], [5], [6].

Traditional MOPSO algorithms assume a homogenous error-free distributed environment [7], [8]. The emergence of grid computing has resulted in circumstances where the available systems are heterogeneous in nature, and may fail during processing. Indeed, it is a commonsense practice to assume distributed computing will experience failures and the different classes of failure have been systematically described [9]. In many cases, however, little attention had been given to the possibility of failure in the design of optimisation algorithms, relying on decentralisation to provide robustness [10].

To verify that nodes returning incorrect answers do have a significant negative impact on the quality of results an experiment was performed to check if a small number of nodes returning incorrect values actually significantly affected the outcome of a MOPSO evaluation. Fig. 1 shows the result of running a simulated MOPSO evaluation on the well-known ZDT2 test function [11] using 100 nodes without any failure, while Fig. 2 shows the result of running a simulated MOPSO evaluation on the concave ZDT2 function using 100 nodes with 1 of the nodes erroneously returning results for the convex ZDT1 function. The damage caused to the attainment surface by even the single failing node is obvious, with large gaps appearing in the attainment surface where erroneous solutions dominate legitimate results and remove them from the archive.

Some previous work has been directed at modifying MOPSO algorithms to make them more suitable for use in heterogeneous distributed environments. The algorithms introduced in Lewis et al. [7] deal in large part with crash failures, omission failures and timing failures by relaxing the process synchronisation in algorithm iteration. However, this still leaves the problem of receiving timely but incorrect responses, response failures, either due to hardware or software failure, or malicious distributed nodes that may deliberately return incorrect data. As such, we wish to find an algorithm suitable for detecting response errors and preventing those errors from being returned as valid solutions.

While different aspects of the general problem of failure and fault tolerance have been studied for many years (see, for example, Lamport [12] and Johnson [13]) much of the published work is systems-oriented, for example outlining desirable qualities for an architecture for dealing with errors in an asynchronous distributed environment [14]. Of course, it is possible to rely on the distributed computing systems to provide mechanisms for dealing with faults, but this cedes control of the issue to another party and, when dealing with...
response failures, this may be insufficient for some sensitive applications. It is also possible to address the issue by masking failures using redundancy, as in some recent investigations of distributed optimisation in unreliable networks [15]. In general, in order to detect and correct response failures a minimum of Triple Modular Redundancy (TMR) must be implemented. Used for many years to provide fault tolerance in electronic circuits (see, for example, Lyons and Vanderkulk [16]) TMR is generally applicable and reliable but inevitably requires replication of computing resources if processing time is not to increase to an undesirable degree.

If the costs of replication are to be avoided, some way of masking failure must be found that does not impose additional computational load. In a parallel, population-based optimisation algorithm, many points in parameter space are iteratively, simultaneously sampled. Assuming some piece-wise linear, quasi-continuous mapping between parameter space and objective space, it may be sufficient to find a means by which solutions within a neighbourhood can be compared to each other, to determine whether a particular result can be considered “reasonable”. This problem of reaching agreement is well-known as the Byzantine agreement problem [12] and its theoretical application to an asynchronous, heterogeneous distributed environment has already been suggested [17], [18]. Byzantine agreement is a method for a group of involved parties to agree on a course of action via unreliable communication and with some of the parties potentially being traitors. Traitors may seek to impede the agreement by means such as sending incorrect values to other involved parties (which makes reaching an agreement on the majority difficult) and/or lying about the values received by them from other parties. It has been shown that it is possible to guarantee that an agreement will be reached if less than 1/3 of the involved parties are traitorous.

However, existing work is based on an environment where the only concern is for the involved parties to agree upon some fixed value rather than evaluating the “correctness” of that value. In addition, participants are assumed always available and a cryptographic dealer must be available at the time the system starts up. These assumptions are problematic in a distributed MOPSO environment because:

1) due to the possibility of churn or the use of peer-to-peer environments, it is not possible to guarantee that all nodes will have access to the dealer at the time the system is started, or even that a node will always have access to the dealer and,

2) each node is tasked with evaluating a distinct point in parameter space and will have different values for a correct answer, making it insufficient to simply compare the values and take a majority vote on the correct value.

As a result an algorithm is needed that:
- allows nodes in the swarm to change,
- determines agreement between nodes based on similarity of values within some threshold decided by how close the nodes are to each other, and
- doesn’t inherently rely on any one node being privileged, while still preserving the agreement between loyal nodes that underpins Byzantine agreement.

II. THE ALGORITHM

This algorithm was developed as a modification of the Byzantine Lieutenants Algorithm [12]. It assumes that the
smallest possible neighbourhood for single error correction, i.e. 4 nodes, is desired for evaluation, as the inaccuracy in judgement of values between nodes is likely to increase rapidly with distance from the node being evaluated. (Extending it to more than 4 nodes is merely a matter of implementing the more generalized form of the Byzantine Lieutenants Algorithm and correspondingly extending the Euclidean distance function.)

Note that when the “value” of a node is used in the following it refers to the set of values that represent the solution’s location in objective space.

The Byzantine Lieutenants’ Algorithm is divided into two parts which assume that there is a single commander (who is sending the message that agreement must be reached on) and several lieutenants (who need to agree on the value the commander sent). The number of traitors that can potentially be detected is \( m \) (in our 4-node neighbourhood implementation \( m = 1 \)), \( n \) is the number of parties involved and it is assumed that all loyal parties have the same agreed-upon default value.

**Algorithm 1** The Byzantine Lieutenants’ Algorithm, \( OM(0) \)

1: The commander sends its value to every lieutenant
2: Each lieutenant uses the value sent from the commander, or a previously agreed upon default value if no value is received.

**Algorithm 2** The Byzantine Lieutenants’ Algorithm, \( OM(m) \), \( m > 0 \)

1: The commander sends its value to every lieutenant
2: For each \( i \) let \( v[i] \) be the value Lieutenant \( i \) receives from the commander, or else be the previously agreed upon default value if no value is received. Lieutenant \( i \) acts as the commander in Algorithm \( OM(m - 1) \) to send the value \( v[i] \) to each of the other \( n - 2 \) lieutenants.
3: For each \( i \) and \( j \neq i \) let \( v[j] \) be the value Lieutenant \( i \) received from Lieutenant \( j \) in step 2 (using Algorithm \( O(m - 1) \)) or else a previously agreed upon default value if no value is received. Lieutenant \( i \) uses the majority of \( v[1],...,v[n - 1] \).

When agreeing upon a node’s value the agreed-upon default value is some value impossible to achieve from the function being evaluated (so that loyal nodes can know to vote FALSE in the following steps if this is reached). When voting on whether or not to insert a value into the archive the agreed-upon default is FALSE, as it is more important to keep poor results out than to let borderline results through.

One round of MOPSO is performed normally. However before any values are added to the archive the following algorithm is performed for each node that has returned a result:

1) The nearest 3 neighbours of the node are found (using Euclidean distance in parameter space as a measure of distance)
2) In a peer-to-peer system the node to be evaluated would perform Byzantine agreement with the 3 nodes in its neighbourhood, on its value.
3) In a Master-Slave system the Master sends out the value to be checked, which is the value the Master received from the node to be evaluated, to the node to be evaluated and its 3 nearest neighbours.
4) After all the nodes in the group have the value to be agreed on, Byzantine agreement is performed once with each node as the initial commander for its iteration. The commander chooses whether it votes yes or no, depending on whether or not the value to be evaluated is within a threshold of its own value, the width of the threshold determined based on the distance between the node to be evaluated and the commander, in parameter space. The exact formula for determining the size of the threshold was determined empirically, and may need to be determined individually for differing optimisation problems.
5) After the majority outcome of the rounds is calculated for each node, in a peer-to-peer system each node would then either include the value in its archive (if the majority of rounds voted yes) or discard the value (if the majority of rounds voted no), in a Master-Slave system the Master uses the majority of the majority vote for archive inclusion.

### III. Computational Experiments

The algorithm was tested using a swarm with a population size of 100 performing a total 3600 evaluations. These values were chosen as they are large enough to represent a reasonably sized pool of nodes each performing a reasonable number of iterations, while also being small enough to limit the simulation time. “Correct” nodes evaluated the ZDT2 test function and voted properly, while erroneous nodes returned results from the ZDT1 test function and voted randomly. These functions were chosen as ZDT1 dominates ZDT2 at almost every point and therefore represents a worst case scenario. Simulations were performed using a Master-Slave setup. For each run two archives were maintained: one to which only solutions that passed the Byzantine agreement were added and one that had all solutions added.

Multiple trials of two possible scenarios were performed:
- Fixed Errors, where a guaranteed number of errors occurred every iteration, or
- Random Errors, where a probability was assigned for each job in an iteration to fail and return incorrect results but recover before the next iteration.

The following figures show the results for these trials with various rates of failure. All used \pm X^2 to determine the size of the threshold, where \( X \) is the Euclidean Distance in parameter space. This value was empirically determined. (Note: the parameters for the ZDT test functions are drawn from the interval \([0,1]\) so the square power is a reductive operator.)
A. Discussion

As can be seen from the figures, the Byzantine agreement does have positive effects for fixed error rates of 1% and 2% (Figs. 3 and 4), and 1% and 2% random error rates (Figs. 6 and 7). For a fixed 1% error rate (Fig. 3), Byzantine agreement has strong results and the recovered attainment surface strongly resembles the expected Pareto-optimal front without any errors. In contrast, without correction the attainment surface approximates the convex Pareto-front of the ZDT1 test function; almost all of the correct results from ZDT2 have been destroyed. The positive results from the use of Byzantine agreement error correction is as expected as, with only 1 error, no group is ever going to have more than one erroneous node, allowing a reliable majority decision to be reached. Any error value accepted would have to be quite close to the correct result. The results for a 1% probability of error (Fig. 6) are worse but still show a positive influence. This is also consistent with what would be expected, since while a fixed 1% error rate guarantees only one error ever being encountered by a group, a 1% error probability allows for the chance of two or more errors in a group, which may allow some ZDT1 values to enter the archive, and removing incorrectly dominated, legitimate solutions. This effect can be seen particularly in the centre of the curve.

For a 2% fixed error rate (Fig. 4), the Pareto-front is retained over much of its extent, but large distortions can be seen where ZDT1 points have slipped through on two occasions, dominating nearby ZDT2 points and removing large numbers of correct solutions from the archive. This results in data being salvageable, if some form of examination is available, but renders the results unsuitable for automated use. For a random 2% error rate (Fig. 7) the result is similar.

Once a 5% error rate has been reached with either fixed or random errors (Figs. 5 and 8), the Byzantine agreement has little beneficial effect on the shape of the resulting final curve. With a 5% error rate, the groups of 4 will contain two or more erroneous nodes frequently enough to allow a number of ZDT1 points to enter the archive. Since ZDT1
results dominate ZDT2 results when they enter the archive, it only takes a few of these to remove large numbers of valid solutions from the archive, which will result in the swarm being moved towards the parameter sub-spaces that present good ZDT1 results, causing the correct nodes to return poor results.

IV. CONCLUSIONS

The modified form of the MOPSO algorithm using Byzantine agreement works to eliminate most errors for a 1% fixed error rate, and produced salvageable data for a 1% random error rate and both fixed and random 2% error rates. As the error rates increase the ability of the algorithm to correct errors and effectively mask failures diminishes. This is to be expected since the small groups used for the Byzantine agreement are easily corrupted if errors happen to occur in closely grouped regions. This detrimental effect may be reduced by increasing the group size.

It should be noted that this error correction was achieved without any additional objective function evaluations, all remediation using solutions already computed. The results obtained, though preliminary in nature, demonstrate a potential method for enhancing the algorithmic fault tolerance of a range of population-based algorithms.

In terms of time efficiency the modified algorithm with Byzantine agreement ran in time comparable to the algorithm without Byzantine agreement for similar swarm sizes and job rates, the difference being less than 10% of the run time. This discrepancy could possibly increase in a real environment, as Byzantine agreement uses several rounds of data exchange which would take time, but it could be anticipated that this would be compensated for by the function evaluation times in a practical environment being much larger than those for the test functions used in these experiments.

In summary, this method could be expected to have positive effects in situations where the error rate is expected to be low and being able to salvage some useable data, even when some corruption occurs, is valuable.
The ZDT2 test function is consistently dominated by the ZDT1 function, making these experiments a worst case scenario for errors. This work only implemented a neighbourhood of 4 (the node to be evaluated and the closest 3 neighbours). Increasing the neighbourhood size could be expected to have a positive effect on the number of errors that can be detected. What upper limit may exist on the size of the neighbourhood that has a positive effect is an open question. In addition, the effect of colluding, malicious nodes may warrant investigation.

REFERENCES