

Reliable Computation Using Unreliable Network Tools

Andrew Hamilton-Wright
Systems Design Engineering
University of Waterloo
andrewhw@ieee.org

Deborah Stacey
Computing and Information Science
University of Guelph
dastacey@uoguelph.ca

1. Introduction

In network tools, reliability is usually approached by attempting to make the network tool reliable. These efforts deliver transparent, fault tolerant network computation engines which internally require a lot of state and global system knowledge in order to ensure the successful progress of computation tasks throughout the system [1].

Here we propose a framework for the design of distributed/parallel systems that are not dependent on reliable networks or global network status information. There is a class of stochastic algorithms that are amenable to decomposition into independent components that have their own (independent) chance of failure. The system, as a whole, can be designed to not manage failures inside of the system. Instead these failures can be more successfully managed in the context of algorithm design.

By dividing the algorithm along lines of problem definition versus atomic calculations to be completed, failure of individual calculations become irrelevant to the convergence of the solution as a whole. Many algorithms can be constructed along these lines. The family of algorithms on which we concentrate are those which use a significant stochastic component in their computation. The *randomness* within the stochastic component will be used to *hide* the failure from the network tool in a way which is non-destructive to the solution generation progress. We use a Genetic Algorithm as an example; the approach can be as easily applied to any Monte-Carlo technique, and to many techniques within Grid Computing. In fact, this design philosophy can be seen as a way to evolve Grid Computing to use the existing Internet without the use of a set of global standards that imply a significant use of bandwidth just for organizational structures.

2. Design Technique

A system designed using this philosophy is divided into two parts; one within the solution generation algorithm, the other forming the network calculation tool [3]. The *Solution*

Manager within the stochastic algorithm is the portion of the system which decides whether enough *solutions* have been generated by the network system to continue to the next iteration.

The overall design will be familiar to anyone who has looked at SETI@Home [2], Folding@home [5], *et cetera*. The intention is to gather computing resources from a large number of machines; where each machine is primarily dedicated to other tasks. The difference is that the task to be distributed does not have to be *embarrassingly* parallel; algorithms that base their future actions on several calculations that are done in parallel can be designed with our technique.

2.1. Network Tool

The network tool is simply a broker process which keeps a list of *worker* machines which have recently (within the last hour) indicated a willingness to take on a task. When a task is assigned, it is simply requested of the next available *worker*, which may refuse the task. A *best-effort* scheme of polling possible workers is in place to find a worker willing to perform the task. The worker negotiates back with the work-generation program directly to acquire and return problem-related data.

No effort has been made in the network component to gracefully handle failure. If any part of the system crashes, or is shut down, the failure is silent. As the failures will be silent, there is no need to keep global state (other than the current list of registered workers). This makes the system extremely scalable – preliminary tests indicate that at least 10000 machines can be registered to a single broker before network traffic becomes an issue.

Due to the silent failure however, the reliability of the system needs to be handled within the generational stochastic algorithm. Using a simple modification of the standard stochastic scheme, this is easily done for a large class of algorithms.

The algorithm which is governing the process and working towards an overall goal (the stochastically based algorithm) needs to have only these properties:

1. **Ability to generate Work:** The problem must be able to be divided up into separable, independently calculable components. This is essentially the definition of a parallel problem. Each of these components will be a unit of *work* performed by the network tool. Any one of these units may or may not be completed in a single *generation*
2. **Stochastic Design:** The stochastic design of the system allows us to treat as *unknown* any units of work for which we do not have a result. For such units, we simply assign either a random probability of usage in any continuing calculations or we use a previous value or we assign a random value; whichever is appropriate for the algorithm.
3. **Complete Problem Knowledge:** The knowledge of the problem must be encoded somewhere in the system; this is the place where the solution is being built from the component parts. This, therefore is where the problem knowledge must reside. This means that this node is critical to the continuing generation of a solution, but no other node in the system needs to be critical.

We will term the portion of the system generating the individual units of work the *Solution Manager*. This component deals in units of work to be done, and thresholds at which it will decide to continue to the next *generation* of the solution.

These thresholds can be based on either patience (the amount of time the user is willing to wait for a solution), or on a threshold based on the number of completed solutions returned, or both. The *patience* plan allows for a rough solution to be calculated in a finite, predetermined period of time. The major limit on this time period is simply the amount of time needed to guarantee some kind of solution – this is problem depended, but can be generated simply by sampling (that is, no complete knowledge of the solution path needs to be known by the researchers). Based on the amount of *knowledge* returned, we can estimate how good the (current) answer is.

Once the threshold has been reached, the Solution Manager indicates that the main calculation can continue. It is possible that later calculations may arrive – these can either be incorporated into the overall answer at a later iteration, or can be ignored.

This design technique has been used to construct a genetic algorithm that trains and tests a set of back-propagation neural networks for the classification of erythrocyte cells in Pap smears. A set of homogeneous machines in the undergraduate laboratories at the University of Guelph [4] was used as the computation resource. This is in every sense a *real* test on an unstable network. The system

was compared to a standard parallel algorithm designed to solve the same problem. The new algorithm outperformed the *robust* algorithm by about a factor of four.

3. Discussion

The benefits of this design are obvious in the decreased run times. The limitations deserve some discussion. In this system, the calculation limit under load will be bandwidth based. The bandwidth is a product of: file size \times # of employed workers which is essentially the amount of data required to define the problem. In other systems, the global overhead must also be included in this calculation – our overhead is a negligible because of the lack of system state. The scalability is at least into the tens of thousands per broker process.

Given an algorithm which is making many, iterative calculations to arrive at a *solution* (optimal or otherwise), it is possible in many instances to define the problem in such a way that progress can be made on the solution without full knowledge of all possible pre-calculations.

Once the drive towards the solution is phrased in this way, it becomes clear that missing or delayed calculations do not necessarily prevent the solution engine from doing valuable work. Our work demonstrates this using genetic algorithms, but is equally valid for other stochastic (Monte-Carlo, Grid, *et cetera*) methods. Recent work by one of the authors has shown that the BP algorithm, which demonstrates a very fine-grained parallelism, can still function effectively with only partial information about weight changes.

The scalability is much better than in conventional parallelism, which should allow us to attack much larger problems. It is our intention to explore the full scalability of this system and the find other stochastic algorithms that are suited to this design technique.

References

- [1] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design*. Addison-Wesley, second edition, 1994.
- [2] D. Gedye and C. Kasnoff. SETI@home project. Internet Website <http://setiathome.ssl.berkeley.edu/project.html>, 1998.
- [3] A. Hamilton-Wright. A working guest: Hosting genetic algorithm computation across multiple networked agents. Master's thesis, University of Guelph, Guelph, Ontario, Feb. 1998.
- [4] A. Hamilton-Wright and D. Stacey. Fault-tolerant network computation of individuals in genetic algorithms. In *2002 Congress on Evolutionary Computation*, pages 1721–1726. IEEE, Feb 2002.
- [5] V. Pande and S. University. Folding@home project. Internet Website <http://folding.stanford.edu>, 1998.