Spread-Spectrum Computation

Derek G. Murray and Steven Hand
Computer Laboratory
University of Cambridge
{Firstname.Lastname}@cl.cam.ac.uk

Abstract

We observe that existing methods for failure-tolerance are inefficient in their use of time, storage and computational resources. We aim to harness the power of idle desktop computers for data-parallel computations, which are particularly sensitive to failure, and propose *spread-spectrum computation* as a suite of techniques to mitigate failures in an internet-scale distributed system.

Spread-spectrum computation will use *computation dispersal algorithms* to add redundancy to computations, in order that they may tolerate a particular failure distribution. In this position paper, we introduce computation dispersal algorithms, providing examples of their implementation and applications.

1 Introduction

Failure-tolerant computing is inefficient. Existing methods waste time by repeating partly-completed tasks [4, 9]; computational resources by sending the same task to several hosts [1]; or storage by taking regular checkpoints [17]. However, machine failures are common [4], and we propose that it is both feasible and more efficient to deal with failures in advance of computation. Therefore, we propose *spread-spectrum computation*, which uses techniques from forward error-correction to add a controllable amount of redundancy to the inputs of an algorithm.

The aim of this work is to enlarge the scope of internet-scale distributed computing. Currently, such approaches are limited to solving *embarrassingly-parallel* problems – i.e. those in which the overall problem can trivially be divided into a large number of parallel tasks, with no dependencies between the tasks [1]. When the success of a computation depends on volunteer-provided resources, failure tolerance is critical: volunteered computers may crash or go offline at any time, and the internet connection may fail. Fortunately, since there is no dependency between the tasks, the simple failure-tolerance mechanisms (retry or send-to-many) work perfectly well.

Unfortunately, not all algorithms are embarrassingly parallel. An important class of problems can be solved using *data-parallel* algorithms, in which many parallel processors cooperate to solve a task by each processing a different piece of the input data [9]. There are dependencies between the processors at the beginning and end, when the data is respectively distributed and collected, and there may be dependencies during execution in order to exchange intermediate data (e.g. a *halo-swap*). Now, if a node fails, it could stall the entire computation.

A workable system must achieve three properties:

Failure tolerance Jobs in the system will run on volunteer-provided computing resources, which may fail or be reclaimed at any time. Therefore, such failure must not impair the progress of any job.

Latency tolerance The volunteer-provided computing resources will be heterogeneous, and will be connected by heterogeneous network links. Therefore, "stragglers" in the system must not cause a bottleneck for any job.

Decentralised management We aim to scale this system to millions of processing nodes. Therefore, it would not be feasible to manage nodes and schedule jobs centrally: we need decentralised management tools.

Here we concentrate on failure tolerance, and provide brief descriptions of the mechanisms for providing latency tolerance and decentralised management in Subsection 3.2.

We propose a spread-spectrum approach, because spread-spectrum techniques have a long history of improving robustness in communication [16] and – more recently – in storage [7]. A typical spread-spectrum approach involves each principal independently spreading

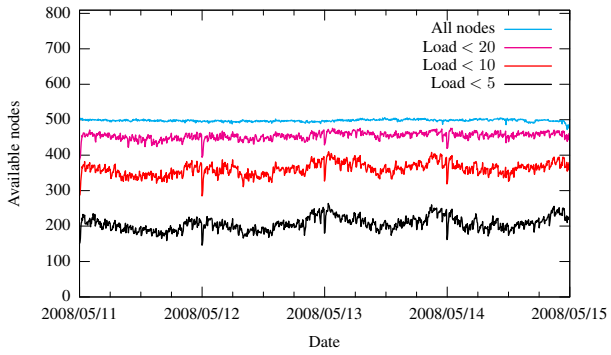

Figure 1: Time series of node availability.

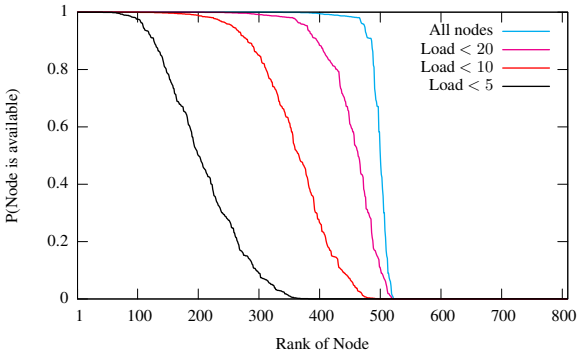

Figure 2: Rank distribution of node availability.

its input across a pseudorandomly selected ensemble of resources – for communication, these would be frequencies; for storage, block addresses. The randomness can provide robustness and security, but some redundancy is required in case two principals randomly choose the same resource.

Spread-spectrum computation is a combination of two ideas: *computation dispersal algorithms* (CDAs) (Subsection 3.1) and *distributed random scheduling* (Subsection 3.2). In this paper, we concentrate more on the redundancy provided by CDAs, but include a discussion of scheduling for completeness. We also provide examples of algorithms to which spread-spectrum computation can be applied.

2 Node failure trace study

It is well accepted that large-scale distributed systems experience failures [4, 9]. While they are online, however, the failing nodes provide a worthwhile computational resource. In this section, we present an analysis of data from the CoMon project, which records the state of all nodes on PlanetLab [11]. Although PlanetLab does not exactly model our target network of idle desktop computers, it represents a large distributed system that is affected by transient node failures and network outages.

In this paper, we are concerned about each node's availability, which CoMon detects by pinging all nodes every five minutes. We also consider load: CoMon reports the five-minute load average of each node. In our system, when a node takes much longer to respond than others, it is considered to have failed, and heavier-loaded nodes will (modulo processor heterogeneity) tend to be slower. In each of our analyses, we differentiate between a node simply being online, and having a load average below a certain threshold (5, 10 and 20, respectively).

We begin by looking at node availability for a period of four days, from the 11^{th} to the 14^{th} of May 2008, inclusive. On these dates, the PlanetLab network comprised

	Number of available nodes		
	Always available	Transiently available	(Effective)
All	339	184	(158.7)
Load < 20	210	311	(244.5)
Load < 10	103	394	(257.9)
Load < 5	42	339	(164.2)

Table 1: Counts of available nodes. "Always available" nodes are nodes for which no failure was detected. "Transiently available" nodes are those that are available some of the time, but have at least one failure. The effective capacity of the transient nodes is obtained by summing the fractional availability for each transiently-available node.

809 nodes. Figure 1 shows how system-wide node availability changes over time. The first thing to note is that the number of online nodes is fairly steady ($\mu = 497.7$, $\sigma = 3.2$). By comparison, looking only at nodes with a load average less than 5, far fewer nodes are available, and the availability shows diurnal variations ($\mu = 206.2$, $\sigma = 19.3$). These variations suggest that the optimal amount of redundancy may be a function of the time of day.

Now that we have established that some nodes do fail, we must show that the set of failing nodes provides a useful computational resource (when those nodes are transiently online). Figure 2 shows a rank distribution of node availability. For each series, we can see that some nodes are always available (where $P = 1$), some nodes are never available ($P = 0$), and some nodes are transiently available ($0 < P < 1$) – these are the nodes that our failure tolerance mechanism attempts to harness. The interesting aspect of Figure 2 is therefore the area under the sloping part of each series.

The benefit of failure tolerance is that it enables the transiently-available nodes to participate usefully in computation. We make the assumption that the effective

computational capacity of a node is directly proportional to the fraction of the time that it is available, if we ignore processor heterogeneity. The effective computational capacity, c, of a set of nodes, S, is defined as:

$$c(S) \quad = \quad \sum_{n \in S} P(n \text{ is available})$$

Table 1 summarises the effective capacity for each scenario that we considered. For example, we see that the effective computational capacity of the 184 nodes that are transiently online (with any load) is equivalent to 158.7 always-on computers. For nodes with a load average less than 10, this rises to 257.9 effective nodes, which is 71% of the total system capacity.

This preliminary study confirms that the transiently-available PlanetLab nodes comprise a substantial proportion of PlanetLab's total computational resource. A further concern is the effect of load on each node: Rhea *et al.* performed a more detailed study, and discovered that the latency of some operations could vary by more than two orders of magnitude [14]. Their solution of adding redundancy to improve performance agrees with our intuition about distributed random scheduling in §3.2.

3 Spread-spectrum computation

Spread-spectrum computation takes a spread-spectrum approach to parallel computation. As stated in the introduction, its three main goals are failure tolerance, latency tolerance and decentralised management. These are fulfilled using two complementary techniques: *computation dispersal algorithms* (which provide failure tolerance), and *distributed random scheduling* (which provides decentralised management and latency tolerance). In this paper, we concentrate on the computation dispersal algorithms (Subsection 3.1), but give a brief description of distributed random scheduling (Subsection 3.2) for completeness.

3.1 Computation dispersal algorithms

Spread-spectrum computation uses a *computation dispersal algorithm* (CDA) to add redundancy to the inputs of an algorithm. It does this by adding *encoding* and *decoding* steps to an existing parallel algorithm (see Figure 3).

We can define a CDA by analogy with Rabin's information dispersal algorithm (IDA), which encodes a file of length L into n pieces, each of length L/m, such that any m pieces are sufficient to reconstruct the original file [13]. In effect, any subset of the pieces with length totalling the original length of the file can be used to reconstruct the original file. An optimal CDA, then, encodes a computational input of size L into n pieces, each of size L/m, such that processing any m pieces is sufficient to obtain the correct result.

Borrowing from earlier work on algorithm-based fault tolerance [8], we can show that an optimal CDA exists for the problem of matrix-vector multiplication. Given $A \in \mathbb{R}^{m \times p}$ and $\boldsymbol{x} \in \mathbb{R}^p$, we might calculate $\boldsymbol{y} = A\boldsymbol{x}$ by distributing each row, $\boldsymbol{a}_i^T$, of A to a processing node i (where $1 \leq i \leq m$), which calculates the dot product $y_i = \boldsymbol{a}_i^T \boldsymbol{x}$.

If we wished to tolerate a single node failure, our CDA could transform A into $A' \in \mathbb{R}^{(m+1) \times p}$ such that:

$$A' \quad = \quad \begin{bmatrix} A \\ \boldsymbol{r}^T \end{bmatrix}$$

$$\text{where } \boldsymbol{r}^T \quad = \quad -\sum_{i=1}^{m} \boldsymbol{a}_i^T$$

It is clear that:

$$A'\boldsymbol{x} = \boldsymbol{y}' = \begin{bmatrix} y'_1 \\ \vdots \\ y'_{m+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{r}^T \boldsymbol{x} \end{bmatrix}$$

To tolerate a single node failure, we distribute each of the $m+1$ rows of A' to processing node i (where $1 \leq i \leq m+1$). If node j fails (where $1 \leq j \leq m$), we can compute the missing y_j by summing together all other y_i values. Chen and Dongarra show that this approach generalises to tolerate k failures, by adding k specially-weighted checksum rows [3].

Obviously, if a CDA is to be useful, it must allow efficient encoding and decoding. Constructing $\boldsymbol{r}^T$ requires $O(mp)$ additions, which is relatively expensive compared to the cost of performing the matrix-vector multiplication. However, many algorithms make it possible to amortise the encoding cost, by reusing the encoded matrix (see §4 for examples). More pertinently though, the k-failure-tolerant encoding procedure requires $O(mk) = O(n^2)$ operations to create the k checksum rows [3].

We therefore propose to use low-density parity-check (LDPC) codes for use in this CDA, as they enable encoding and decoding in $O(n)$ operations [15], which allows scaling to much larger problem instances. An additional benefit of LDPC codes for this CDA is that the checksum rows retain the sparseness of the data rows, unlike the single-row example, which would generate a dense $\boldsymbol{r}^T$, even for sparse A. One disadvantage is that the LDPC code introduces some overhead: if we begin with m partial inputs, we require mf ($f \geq 1$) encoded partial outputs to obtain the correct result. However, $f \to 1$ as $m \to \infty$ (and as $m/n \to 1$), and Plank and Thomason

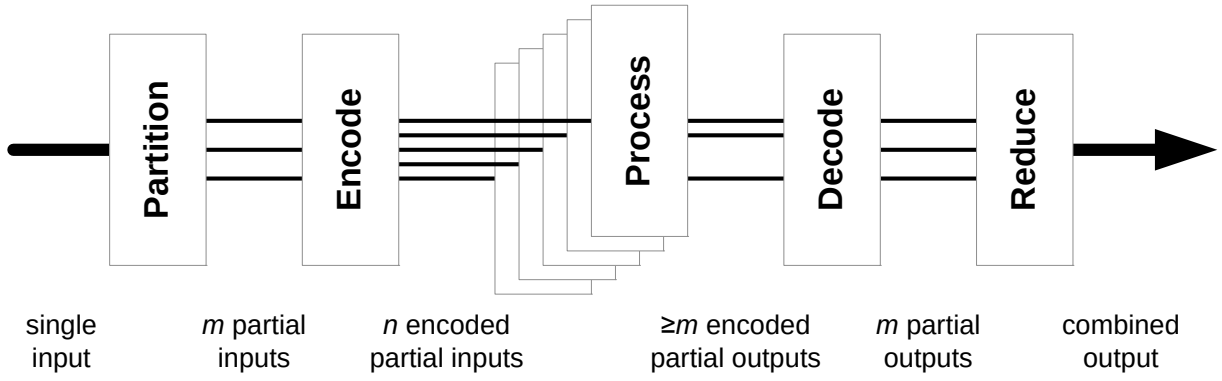

Figure 3: The data flow in a data parallel algorithm when a CDA is used. The CDA is responsible for the "Encode" and "Decode" steps.

have observed that the overhead can be made less than 5% (for $m = 1000$ and $n = 1500$) [12].

We have shown an example CDA for matrix-vector multiplication, but does this approach generalise? The checksum scheme adapts naturally to all linear transformations (i.e. those that satisfy the superposition property), which includes many other matrix operations and signal processing techniques. Huang and Abraham adapted algorithms for matrix multiplication and LU decomposition [8], which could lead to other CDAs. The search for further CDAs is the subject of our ongoing research.

3.2 Distributed random scheduling

Distributed random scheduling works in combination with computation dispersal algorithms to provide efficient, scalable and load-balanced scheduling across an internet-scale pool of idle desktop computers. In an internet-scale distributed system, it is not feasible to have a central scheduler that allocates parallel jobs to processing nodes. Any such scheduler would have to monitor the state of all processing nodes, and keep this information up-to-date. The use of redundancy gives us more flexibility in our scheduling mechanism.

Under distributed random scheduling, when a submitting node wishes to submit a job to the system, it selects n processing nodes at random, and sends the encoded partial inputs to these nodes. This provides two attractive properties. Firstly, if all submitters choose processing nodes at random, the overall system load will be balanced, which implies efficient resource utilisation. Moreover, in a heterogeneous system using an m-out-of-n CDA for failure tolerance, the performance bottleneck will be the $(n - m + 1)^{\text{th}}$ slowest node, rather than the slowest node, which improves latency tolerance and hence performance. This concurs with previous observations that individual "stragglers" in a distributed system greatly affect the performance of global computation [4, 14]. The CDA allows our system to produce a result after m responses, so the slowest $(n - m)$ nodes do not delay the result.

Obviously, if it is infeasible for a central node to maintain state information about all nodes, it is even more infeasible to expect each submitter to maintain such a list. We can achieve random selection by structuring the processing nodes as an expander graph, and performing a random walk on the resulting graph [10].

4 Potential applications

The best applications for the techniques described above will be those which require a large amount of data to be processed, such that it is infeasible to store the entire data set on a single processing node.

Initially, we are investigating large scale information retrieval tasks, such as PageRank calculation [2] and latent semantic indexing [5]. The PageRank of a collection of web pages may be computed as the principal eigenvector of the modified web page adjacency matrix, while latent semantic indexing involves calculating the singular value decomposition of the term-document matrix.

Calculating the principal eigenvector of a matrix can be achieved by power iteration [6]. The computationally-intensive part of this calculation is a repeated matrix-vector multiplication, for which we presented a CDA in Subsection 3.1. When used with web graphs, which form sparse matrices, the LDPC-based encoding scheme would be particularly appropriate. The singular value decomposition and full eigendecompositions can be calculated using Lanczos iteration [6], which is a more sophisticated version of this algorithm, but which is still based on iterative matrix-vector multiplication.

We are considering several other applications. The

above CDA could also apply to boundary value problems, which may be solved iteratively using the conjugate gradient method. An adapted version of the CDA could also apply to multi-dimensional Fourier transforms, which have various applications in image analysis and molecular dynamics, amongst other areas.

5 Related work

In the volunteer-computing space, this work compares most directly with BOINC [1] and Condor [17], which represent different approaches to harnessing idle desktop computers. BOINC (on which projects such as SETI@Home are based) uses a task-farming approach to solving embarrassingly-parallel problems. Since every task is independent, it deals with failures simply by sending out multiple copies of each task to different processing nodes. Condor monitors when nodes in a local network become idle, and performs matchmaking between submitters and idle nodes. Wide-area distribution is made possible by "flocking". Condor supports process-level checkpointing for failure tolerance, although it does not generate consistent distributed snapshots for nodes executing in parallel. Moreover, although it has a rudimentary (failure intolerant) coscheduling feature, this is limited to the local network.

The data-parallel aspect of this work compares with MapReduce [4] and Dryad [9]. These respectively use functional programming and dataflow graph abstractions to process large-scale data sets. Both rely on re-execution as a failure tolerance method, which wastes wall-clock time, especially if tasks are long-lived. Furthermore, they are designed to work in large data centres, but not internet-scale distributed systems, so they rely on a centralised scheduler.

Our proposed system builds on algorithm-based fault tolerance (ABFT), which is intended for large-scale parallel computation [8]. ABFT introduced the idea of encoding the inputs to a parallel computation and computing with the encoded data, and our example CDA is based on the checksumming procedure that its authors describe. The focus of ABFT research has shifted to networks of workstations, and more advanced codes have been presented recently [3]. We intend to develop this work further by experimenting with more-efficient low-density codes, and deploying our system in a widely-distributed setting.

6 Conclusion

We have presented spread-spectrum computation: a set of novel techniques that aim to make data-parallel processing feasible on idle desktop computers. We plan to build and deploy our system in various widely-distributed settings, in order to investigate its real-world performance. In addition, we plan to study the failure characteristics of large-scale distributed systems, in order to model the appropriate redundancy parameters for efficient use of the system. Together, these projects will bring us to a point where idle-desktop computers can efficiently and dependably be used for data-parallel computation.

Acknowledgments

Thanks are due to our colleagues Jon Crowcroft, Theodore Hong, Tim Moreton, Henry Robinson, Amitabha Roy and Mark Williamson for their comments on earlier drafts of this paper. We would also like to thank the anonymous reviewers for their constructive comments and suggestions.

References

[1] ANDERSON, D. P. BOINC: A system for public-resource computing and storage. In *Grid '04: Proceedings of the 5th international workshop on Grid Computing* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 4–10.

[2] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual Web search engine. In *WWW7: Proceedings of the 7th international conference on the World Wide Web* (Amsterdam, The Netherlands, 1998), Elsevier Science Publishers B. V., pp. 107–117.

[3] CHEN, Z., AND DONGARRA, J. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *IPDPS '06: Proceedings of the 20th International Parallel & Distributed Processing Symposium* (Washington, DC, USA, 2006), IEEE Computer Society, p. 76.

[4] DEAN, J., AND GHEMAWAT, S. MapReduce: simplified data processing on large clusters. In *OSDI '04: Proceedings of the 6th symposium on Operating Systems Design & Implementation* (Berkeley, CA, USA, 2004), USENIX Association, pp. 137–150.

[5] DEERWESTER, S., DUMAIS, S. T., FURNAS, G. W., LANDAUER, T. K., AND HARSHMAN, R. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science 41*, 6 (1990), 391–407.

[6] GOLUB, G. H., AND LOAN, C. F. V. *Matrix Computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[7] HAND, S., AND ROSCOE, T. Spread spectrum storage with Mnemosyne. In *Future Directions in Distributed Computing* (Berlin, Germany, 2003), Springer-Verlag, pp. 148–152.

[8] HUANG, K.-H., AND ABRAHAM, J. A. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers 33*, 6 (1984), 518–528.

[9] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd European Conference on Computer Systems* (New York, NY, USA, 2007), ACM, pp. 59–72.

[10] LAW, C., AND SIU, K.-Y. Distributed construction of random expander networks. In *Infocom '03: Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies* (Washington, DC, USA, 2003), vol. 3, IEEE Computer Society, pp. 2133–2143.

[11] PARK, K., AND PAI, V. S. CoMon: A mostly-scalable monitoring system for PlanetLab. *Operating Systems Review 40*, 1 (January 2006), 65–74.

[12] PLANK, J. S., AND THOMAN, M. G. A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In *DSN '04: Proceedings of the international conference on Dependable Systems and Networks* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 115–124.

[13] RABIN, M. O. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36*, 2 (1989), 335–348.

[14] RHEA, S., CHUN, B.-G., KUBIATOWICZ, J., AND SHENKER, S. Fixing the embarrassing slowness of OpenDHT on PlanetLab. In *WORLDS '05: Proceedings of the 2nd Workshop on Real, Large Distributed Systems* (Berkeley, CA, USA, 2005), USENIX Association.

[15] RICHARDSON, T. J., AND URBANKE, R. L. Efficient Encoding of Low-Density Parity-Check Codes. *IEEE Transactions on Information Theory 47*, 2 (2001), 638–656.

[16] TESLA, N. US Patent 723188: Method of Signalling, Mar. 1903.

[17] THAIN, D., TANNENBAUM, T., AND LIVNY, M. Distributed computing in practice: the Condor experience. *Concurrency: Practice and Experience 17*, 2–4 (2005), 323–356.