SANDIA REPORT

SAND2010-6392
Unlimited Release
Printed September 2010

Peer-to-Peer Architectures for Exascale Computing: LDRD Final Report

Jackson R. Mayo, Yevgeniy Vorobeychik, Robert C. Armstrong, Ronald G. Minnich, Don W. Rudish

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2010-6392
Unlimited Release
Printed September 2010

Peer-to-Peer Architectures for Exascale Computing: LDRD Final Report

Jackson R. Mayo Yevgeniy Vorobeychik
Scalable Modeling & Analysis Systems

Robert C. Armstrong Ronald G. Minnich Don W. Rudish
Scalable & Secure Systems Research

Sandia National Laboratories, P.O. Box 969, Livermore, CA 94551-0969

Abstract

The goal of this research was to investigate the potential for employing dynamic, decentralized software architectures to achieve reliability in future high-performance computing platforms. These architectures, inspired by peer-to-peer networks such as botnets that already scale to millions of unreliable nodes, hold promise for enabling scientific applications to run usefully on next-generation exascale platforms ($\sim 10^{18}$ operations per second). Traditional parallel programming techniques suffer rapid deterioration of performance scaling with growing platform size, as the work of coping with increasingly frequent failures dominates over useful computation. Our studies suggest that new architectures, in which failures are treated as ubiquitous and their effects are considered as simply another controllable source of error in a scientific computation, can remove such obstacles to exascale computing for certain applications. We have developed a simulation framework, as well as a preliminary implementation in a large-scale emulation environment, for exploration of these "fault-oblivious computing" approaches.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

High-performance computing (HPC) faces a fundamental problem of increasing total component failure rates due to increasing system sizes, which threaten to degrade system reliability to an unusable level by the time the exascale range is reached ($\sim 10^{18}$ operations per second, requiring of order millions of processors). As computer scientists seek a way to scale system software for next-generation exascale machines, it is worth considering peer-to-peer (P2P) architectures that are already capable of supporting 10^6–10^7 unreliable nodes. Exascale platforms will require a different way of looking at systems and software because the machine will likely not be available in its entirety for a meaningful execution time. Realistic estimates of failure rates range from a few times per day to more than once per hour for these platforms.

P2P architectures give us a starting point for crafting applications and system software for exascale. In the context of the Internet, P2P applications (e.g., file sharing, botnets) have already solved this problem for 10^6–10^7 nodes. Usually based on a fractal distributed hash table structure, these systems have proven robust in practice to constant and unpredictable outages, failures, and even subversion. For example, a recent estimate of botnet turnover (i.e., the number of machines leaving and joining) is about 11% per week. Nonetheless, P2P networks remain effective despite these failures: The Conficker botnet has grown to $\sim 5 \times 10^6$ peers [13]. Unlike today's system software and applications, those for next-generation exascale machines cannot assume a static structure and, to be scalable over millions of nodes, must be decentralized. P2P architectures achieve both, and provide a promising model for "fault-oblivious computing".

1.2 Research Goals

This project aimed to study the dynamics of P2P networks in the context of a design for exascale systems and applications. Having no single point of failure, the most successful P2P architectures are adaptive and self-organizing. While there has been some previous work applying P2P to message passing [7], little attention has been previously paid to the tightly coupled exascale domain. Typically, the per-node footprint of P2P systems is small, making them ideal for HPC use. The implementation on each peer node cooperates *en masse* to "heal" disruptions rather than relying on a controlling “master” node.

Understanding this cooperative behavior from a complex systems viewpoint is essential to predicting useful environments for the inextricably unreliable exascale platforms of the future. We sought to obtain theoretical insight into the stability and large-scale behavior of candidate architectures, and to work toward leveraging Sandia’s Emulytics platform to test promising candidates in a realistic (ultimately $\geq 10^7$ nodes) setting.

Our primary example applications are drawn from linear algebra: a Jacobi relaxation solver for the heat equation, and the closely related technique of value iteration in optimization. We aimed to apply P2P concepts in designing implementations capable of surviving an unreliable machine of 10^6 nodes.

Chapter 2

Example: Emergent Behavior in Botnets

The material in this chapter, which is excerpted from an earlier report [11], provides further explanation and motivation for the use of botnets as an exemplar of emergent robustness that can inform exascale computing.

2.1 Botnet Dynamics

All known early and current peer-to-peer botnets have been built on the Kademlia [9] P2P sharing algorithm. Providing a binary fractal structure illustrated in Figure 2.1, Kademlia defines concepts like "nearest neighbor" and a distance measure between peers. Interestingly, this distance measure is completely aloof from the the physical location or subnet in which the peer is located. This is accomplished by generating a random 128-bit hash key that will almost certainly be unique in the bot-world and then determining the position of the peer in Kademlia space from there. The implementation of Kademlia most botnets use, called Overnet, provides the connectivity of the Kademlia algorithm as a protocol plus a means for bootstrapping newly infected nodes into the net.

The robust functioning of a botnet depends primarily on maintaining connectivity and coordination among infected nodes. The topological aspect of this – understanding the extent of connected clusters in various graphs – is a well-studied problem in mathematical physics known as percolation. In fact, when the amount of local connectivity among "marked" (infected) nodes in a graph approaches the threshold at which very large connected clusters appear (the percolation threshold), the resulting topology is generically self-similar and can be understood using renormalization-group techniques. This provides a particularly simple and relevant example of critical behavior and associated scaling laws.

While there exists somewhere a bot-herder that exerts control over his botnet, it is in the bot-herder's advantage to make the bots as autonomous as possible. Because he does not have to attend to the bots personally, the botnet scales to enormous proportions. It follows that a useful and not too idealized model of a botnet is an array of automata. Each bot is a automaton in the array and has some pre-defined role; the array taken as a whole will exhibit an emergent behavior dependent upon, but not necessarily predictable from, the local behavior. This last observation merits some exploration and is at the crux of the reason to model botnets in the first place.

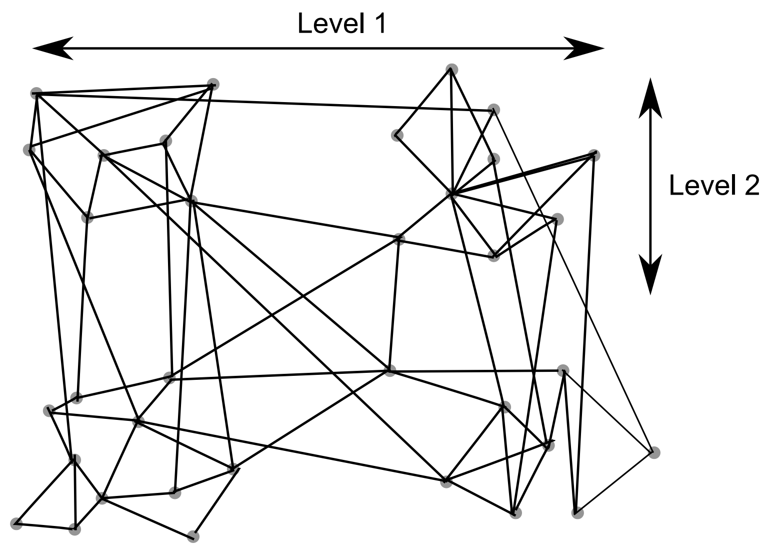

Figure 2.1. Schematic fractal network structure of a botnet using the Kademlia protocol.

Why must botnets be simulated in aggregate and at scale? Turing's halting problem, Rice's undecidability theorem and Gödel's incompleteness proof all state that the emergent behavior of an infinite array of automata cannot be decided ahead of letting it "run". Another way of stating this observation is that, in general, the behavior of such arrays is "irreducible": No simpler or more compact description of the system can be derived. Unlike in statistical thermodynamic systems, there is no bound that can be put on the behavior even probabilistically. Understanding the behavior of large arrays of automata is essential to understanding the behavior of botnets: From a simulation perspective, botnets are little else.

As described in Section 2.2, there is ample evidence for this irreducibility manifested in other arrays of automata – for example, the sandpile experiment in cellular automata [2] and other classical observations [10, 16]. Thus, in the general case, we need to "run" a botnet at scale before we can understand its emergent behavior.

2.2 Complex System Models

Botnets and similar large-scale networks are prototypical examples of "complex" systems, which are characterized by emergent behavior that is irreducible and not predictable *a priori*. Idealized models are a useful tool for understanding complex system dynamics. Cellular automata provide an especially simple setting to illustrate the emergence of rich phenomena from basic underlying rules. Extensive theoretical and computational results have been previously obtained for cellular automata, showing that these systems exhibit a wide range of behaviors seen in the natural and manmade world [16].

A cellular automaton consists of a lattice of cells, each of which carries a definite state at any given time. The evolution of the system is carried out in discrete timesteps. As a result, a specification of the underlying dynamics of the system can be exactly reproduced in a computer simulation, provided enough memory and processing time are available. The lattice of cells can exist in a "space" of one, two, three, or more dimensions. The procedure for "updating" a cellular automaton (evolving to the next discrete timestep) is usually specified via a function that determines the new state of a given cell based on the current state of that cell and its nearest neighbors.

A well-known cellular automaton that provides an instructive comparison for malware is the Bak–Tang–Wiesenfeld (BTW) sandpile model [2], which is defined on a two-dimensional square lattice. This model represents an idealization of the complex behavior of a pile of sand, which becomes unstable when its height exceeds a critical value. In the updating rule, a cell whose "sand level" exceeds the threshold will relax by distributing sand to its nearest neighbors – potentially causing them in turn to exceed the threshold. As a result, if sand is randomly added to a pile in various locations, "avalanches" eventually occur. Depending on the exact configuration at the location and time of the perturbation, an avalanche may be localized or it may sweep over a large part of the system. If this model is run for a sufficient period of time, what is observed is something similar to a second-order phase transition, where avalanches occur on all scales available to the system, obeying a power-law distribution but appearing otherwise random.

The network analogue would be a possibly unremarkable protocol where each machine is similarly arranged on a logical grid and has a counter that is incremented when either a random event occurs or a neighbor communicates with it. If the counter reaches a specific threshold, then the machine will communicate with its nearest neighbors. Because this behavior is isomorphic to the sandpile model, this innocuous-seeming protocol will result in similar communications "avalanches" that will occur at all scales of the participating machines, including the entire network. Such potentially disruptive avalanches are not "directed" in any way but are an artifact of the emergent behavior of the protocol that each participant identically adopts.

Chapter 3

Concepts for Fault-Oblivious Scientific Computing

3.1 Coping with Ubiquitous Failure

Traditional parallel programming techniques have been developed at platform scales small enough that the various causes of component failure (including hardware, operating system, and application faults) can be neglected to a first approximation. Thus the emphasis has been on optimizing the processing and communication performance of applications under a rare-failure assumption, and separately on ensuring sufficient levels of component reliability to comply with this assumption. In common parallel programming models such as MPI, an entire parallel job halts if a single component contributing to the job fails, and so an add-on mechanism is required to deal with failures if one or more of them are likely to occur during a job. The "checkpoint-restart" solution, where a snapshot of the global state of the application is periodically saved to disk and used to resume the job in the event of failure, enables long jobs to run to completion at a manageable cost in additional computing time, but only when failures are sufficiently rare – more precisely, when the system-wide mean time between failures (MTBF) is large compared to the time required to save a checkpoint [5].

Continuing large increases in system size, without corresponding increases in MTBF for individual components (which would require infeasible levels of reliability), are reducing the system-wide MTBF for leading HPC platforms to levels where checkpoint-restart will not remain a practical solution. As the system-wide MTBF becomes smaller than the checkpointing time, forward progress of traditionally designed applications will slow dramatically and their useful completion will become effectively impossible; thus the benefits of exascale computing will not be fully realized in such a framework. Various enhancements, such as predictively monitoring components to checkpoint selectively when failure is more likely [3] and making the checkpoint process itself more efficient, can extend the feasibility of checkpoint-restart to a degree, but by themselves are expected to be insufficient for exascale computing.

A class of solutions with the potential to extend useful HPC scaling to exascale and beyond are those that remove the underlying assumption that failures are rare, and allow jobs to recover from failure via mechanisms more flexible and efficient than checkpointing. Mechanisms of this sort are suggested by the behavior of botnets. There are, however, important differences between botnets and HPC applications: The latter deal with large amounts of data, require high-bandwidth communications, and deliver numerical results whose accuracy is relied upon. Thus, whereas recovery of a botnet from the failure (e.g., disinfection) of some bot nodes consists simply of recruitment of new bot nodes and a gradual, distributed determination of their places in the botnet topology, an HPC application must ensure that the chunk of data located on a failed node is adequately recovered for subsequent computations that rely on it. There arises a distinction, discussed more fully in Chapter 4, between (1) applications that require exactly reproducible results, so that the results obtained upon recovery must be the same as if the failure had not occurred, and (2) applications where error analysis is a normal part of interpreting results, so that alterations in results due to failure and recovery are acceptable if they are sufficiently small and well characterized.

For case (1), the need to recover data exactly is similar to the goal of traditional checkpoint-restart. The key difference is the ability to supplement or replace checkpointing with reliance on locally available intermediate results (in the vicinity of the failed node) to efficiently reconstruct lost data. Recent Sandia work on data-driven parallel programming models, using dependency graphs to identify the inputs truly needed for each computational task, provides a starting point for this. Not only does the elimination of artificial dependencies (such as global synchronization) improve performance even in the absence of failure by allowing more overlap of tasks, but it also has the potential to streamline failure recovery by limiting the amount of data needed to restart a task.

For case (2), even greater efficiencies are possible from "substituting" numerically close and more readily reconstructible data for the lost data. In this approach, errors introduced by failure recovery must be considered as part of the overall error budget of a computation. Because there are several other common sources of error in computational science (floating-point error, discretization error, statistical uncertainty, etc.), substitution error may in fact be negligible for a particular application if it is dominated by another source. The effect of substitution error on results is naturally influenced by the specific error-propagation characteristics of the computation.

A common aspect of both cases is that the detailed behavior of the fault-recovery mechanism depends strongly on the application, in contrast to checkpoint-restart. This implies that fault-oblivious computing will generally require programming languages or annotations that allow programmers to express the needed information.

3.2 Example Applications

The most promising HPC applications for initial validation of our concepts are those that share structural features with botnets, allowing ready generalization of P2P mechanisms, and that exhibit highly robust convergence behavior (damping of perturbations), allowing data substitutions with controllable effects on error. The sandpile model described in Section 2.2 directly motivates our identification of more "scientific" example problems with similar behavior.

First, interactions with nearest neighbors on a lattice (or more generally, with nearby cells on some kind of mesh) are typical of discretizations of partial differential equations (PDEs). An elementary and widely applicable PDE that has a natural nearest-neighbor discretization, and strongly damps perturbations, is the heat equation (written in two dimensions)

$$\frac{\partial \theta}{\partial t} = \kappa \nabla^2 \theta \equiv \kappa \left(\frac{\partial^2 \theta}{\partial x^2} + \frac{\partial^2 \theta}{\partial y^2} \right) \qquad \text{for } (x,y) \in D, \tag{3.2.1}$$

$$\theta(x,y,0) = \Theta(x,y) \qquad \text{for } (x,y) \in D, \tag{3.2.2}$$

$$\theta(x,y,t) = \Theta(x,y) \qquad \text{for } (x,y) \in \partial D. \tag{3.2.3}$$

Here κ is a positive constant, and we have written an initial-boundary-value problem with a Dirichlet condition on the boundary ∂D of the domain D. This can be interpreted physically as the evolution of the temperature field θ in a conductive medium such as a metallic plate, given an initial temperature field and a prescribed temperature profile maintained around the edge of the plate.

A straightforward (and stable) discretization with a particular choice of timestep leads to the Jacobi relaxation method [14], given by the cellular automaton rule

$$\theta_{i,j,t+1} = \tfrac{1}{4}(\theta_{i+1,j,t} + \theta_{i-1,j,t} + \theta_{i,j+1,t} + \theta_{i,j-1,t}). \tag{3.2.4}$$

That is, each cell's value is updated to the average of its nearest neighbors (except for boundary cells, which are maintained at their prescribed values). If the transient behavior of the field is not of interest, then this updating rule can also be considered as simply a route to obtaining the steady-state solution given by the Laplace equation

$$\nabla^2 \theta = 0 \qquad \text{for } (x,y) \in D, \tag{3.2.5}$$

$$\theta(x,y) = \Theta(x,y) \qquad \text{for } (x,y) \in \partial D. \tag{3.2.6}$$

Upon failure of a computational node responsible for some (small) part of the domain, there is a physically natural remedy in this problem: Adjacent cells (i,j) that lack neighbor information substitute their own values $\theta_{i,j}$ for the missing neighbor when updating. This corresponds directly to a Neumann boundary condition (zero heat flux) on the edge of the missing part of the domain, which can thus be interpreted simply as a "hole" in the plate. The effect of this substitution with a small hole is relatively minor because heat can flow *around* the hole. Due to the diffusive nature of the heat equation, small-scale perturbations are strongly damped; thus there is no unstable, growing error from the substitution, and the large-scale equilibration of the temperature field is only slightly altered. Note that Jacobi relaxation – on conventional architectures – is known to be a relatively inefficient algorithm compared to, e.g., multigrid methods. However, the tradeoffs for evaluating algorithms become different in approaching the exascale domain where fault obliviousness is necessary. There, the simplicity and resilience of the P2P-like Jacobi algorithm are powerful advantages.

The structure and behavior of the heat equation are ideal for our P2P approach. Other PDEs used in scientific computing applications lack its strong damping properties, but may have compensating features. For example, the Navier–Stokes equation for high-Reynolds-number turbulence exhibits chaotic behavior in which small perturbations cause a rapidly growing error in the solution. But, because this property applies even to unavoidable roundoff and discretization errors, it is already accepted that the results of Navier–Stokes simulations should be interpreted as random fields from a statistical ensemble. Thus the criterion in developing a P2P-like algorithm for such equations is that the substitutions should have a small effect at the level of statistical properties rather than individual solutions.

Our second example application is closely related in form but motivated by optimization (specifically, dynamic programming) rather than PDEs: the Pre-Jacobi algorithm for value iteration [1]. Value iteration is one of the algorithms for computing the values of states of a discounted Markov decision process (MDP) [15]. An MDP evolves over a sequence of discrete time steps, with evolution proceeding through a series of *states*. At a fixed time step t, the decision maker finds himself in some state $s_t \in S$, and must choose some action $a_t \in A(s_t)$, where $A(s_t)$ is the set of feasible actions in state s_t. Upon taking action a_t in state s_t, he receives a reward r_t and probabilistically transitions to next state s_{t+1}. The Markovian property ensures that both the distribution over rewards and the distribution over next states depend only on the current state and action. We consider a slightly simplified model in which rewards are a deterministic function of current state; thus, $r_t = r(s_t)$. We denote the distribution over next states by P, with $P^a_{ss'}$ meaning the probability of transitioning from state s to s' if action a is taken in state s.

A solution to a MDP is a *policy*, which determines the sequence of action choices as a function of state, denoted $\pi(s)$. An *optimal* policy π^* has the property that it is the policy that maximizes the expected reward with the discount factor δ, that is,

$$\pi^* = \arg\max_{\pi} \sum_{t=0}^{\infty} \delta^t E[r(s_t)|\pi]. \tag{3.2.7}$$

An equivalent way of expressing the maximal expected reward (*value*) of any state $s \in S$ is the Bellman equation [15]:

$$V^*(s) = \max_{a\in A(s)} [r(s) + \delta \sum_{s'} P^a_{ss'} V^*(s')] = r(s) + \delta \max_{a\in A(s)} \sum_{s'} P^a_{ss'} V^*(s'). \tag{3.2.8}$$

The optimal policy can then be computed as

$$\pi^*(s) = \arg\max_{a\in A(s)} \sum_{s'} P^a_{ss'} V^*(s'). \tag{3.2.9}$$

Value iteration [1, 15] is a natural algorithm for iteratively computing optimal state values V^* that stems directly from the Bellman equation. In iteration n, the value $V_{n+1}(s)$ is computed as

$$V_{n+1}(s) = r(s) + \delta \max_{a\in A(s)} \sum_{s'} P^a_{ss'} V_n(s'). \tag{3.2.10}$$

This algorithm is provably convergent to the true vector of valuations V^*. We observe that for the special case of states on a square lattice, with discount factor $\delta = 1$, with a single action that moves to any nearest-neighbor state with equal probability $\frac{1}{4}$ (random walk), and with rewards equal to zero except on the boundary of the domain, this algorithm is equivalent to the Jacobi relaxation method (3.2.4).

Moving from a serial to a parallel implementation of the value iteration algorithm is rather natural. For simplicity, suppose that a processor is tasked with computing a value for a single state $s \in S$ (thereby eschewing any state partitioning issues for the moment). The internal data on the processor will be $r(s)$, δ, and the part of $P_{ss'}^{a}$ with s as the initial state, which can be represented as a matrix with a as rows, s' as columns, and corresponding transition probabilities as entries. At each iteration, the entire vector V_n must be broadcast to all processors in order to compute V_{n+1} [1]. There are three problems with this approach:

1. It requires synchronization, with the concomitant performance hit.
2. It requires broadcast transmission of the entire vector V_n in every iteration n.
3. It requires storing internally the data matrix of size $|S||A(s)|$.

The least of these issues is perhaps the last, although it may become important in practice if the state space becomes extremely large; in any case, any economy here will have direct impact on algorithm performance. The most important issue is the second, since message passing would likely be the performance bottleneck and, with large state spaces, the parallel algorithm will become entirely impractical as a consequence (message passing will dominate computation). Synchronization is the easiest issue to overcome in a natural way: Rather than requiring all states to broadcast their values V_n synchronously, let them do so every time a new value is computed. In this way, each state has the latest vector of values, but the process no longer requires any synchronization. It turns out that asynchrony does not preclude convergence, as long as processors do not stall (i.e., values of all states keep being updated) [8]. The other two issues are resolved by the decentralized *dependency graph*, which we discuss in the next chapter.

Chapter 4

DAS Architecture for Dynamic Task Replication and Substitution

The goal of this chapter is to introduce a general architecture that allows effective failure management and, under some conditions, fault obliviousness in exascale computing scenarios. Here, we envision jobs that are divided into a very large number of interdependent tasks. Consequently, a failure of one processor running a certain task can potentially bring down the entire job due to the intricate interdependencies. Our goal is to manage such interdependencies in order to dynamically restart tasks or replace them with other tasks that generate similar data. To this end, we introduce two graphical data structures: a *dependency graph* and a *substitution graph*. The former keeps track of data dependencies between tasks, while the latter is a representation of substitution relationships between tasks (that is, whether data from one task can substitute for the data from another, possibly failed, task). The former structure primarily manages failures, while the latter is a means of fault obliviousness, insofar as it can be achieved. As the architecture depends on these two graphical structures, we term it the *DAS* (dependency and substitution) architecture.

4.1 Dependency Graph

A *dependency graph* is a directed graph representation of data dependencies between tasks, with each node representing a computational task (e.g., computing the value of a state s), while a directed edge from i to j means that task i depends on task j (for example, a non-zero probability transition from s to s'). See Figure 4.1. In parallel computation we would wish to keep as much decentralization of information as possible, and so it would be greatly undesirable (if not entirely impractical) to maintain a completely centralized dependency graph. Instead, each node maintains the dependency subgraph immediately relevant to it. In our current implementation, a node merely maintains the list of all nodes j that it depends on, as well as those nodes that depend on it, although a more robust implementation would allow it also to maintain dependencies of its inputs (i.e., of the nodes it depends on), etc. The first thing that a decentralized dependency graph wins us is a simple resolution of the scalability issues outlined in Section 3.2 in the value iteration discussion (but applicable broadly). In our case, rather than broadcast messages sent to all nodes, a node queries all its dependencies for their values by the means of message passing (e.g., using MPI). This illustrates that the use of message passing for communication does not entail adoption of the traditional MPI programming model. Perhaps a better implementation would have a node multi-

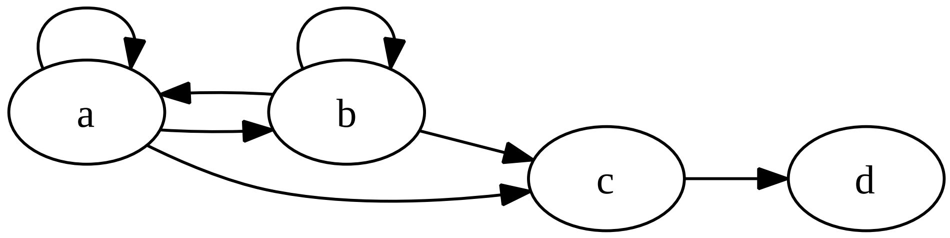

Figure 4.1. Example of a simple dependency graph involving four tasks.

cast results of its computation to all those who depend on it (in addition to responding to standard data requests). In any case, communication is limited to a potentially small fraction of the entire state space. An additional, relatively minor, benefit to a parallel value iteration implementation is that internal storage requires maintaining only nonzero state transition probabilities. In fact, the dependency graph in the Markov decision process (MDP) context is entirely determined by the directed links corresponding to nonzero transition probabilities. In principle, a dependency graph is a completely general architecture, since it certainly allows all tasks to be interdependent, but its benefits only arise if it is sparse (or, more accurately, if the dependencies between processors once tasks are allocated to these are sparse; more on that later).

Focusing now on our stated main task of fault tolerance and obliviousness, the dependency graph provides our main mechanism for fault tolerance. Suppose that a task i detects that one of its dependencies j has failed (perhaps because j did not respond to some query within a timeout period, or i was notified somehow of this failure either by the failing node or by the operating system). If i already has required data from j, it needs to do nothing. If not, it can notify the system of the failure. If there is a provision in the running job to restart specified tasks, or if there is a checkpoint that we can refer back to, the dependency can be placed on the run queue to be restarted so as to provide the required data. Note that we are not requiring that the dependency graph be specified at the time that the main job is started; it can be generated dynamically, as tasks are spawned by the main process. An important aspect is the decentralization of the graph, so that tasks themselves may determine whether any failed dependency actually needs restarting. Consider, for example, a situation in which a task that fails is not depended upon by any other task. Its failure will then go essentially unnoticed, except, perhaps, by the system, and so the job may well continue running, entirely oblivious to any failure having occurred. This ensures that progress can be made even under considerable failures (as compared to the common alternative, which is a halt to the entire system due to occasional isolated failures).

Another utility of the dependency graph is that it can be a means of restoring state without frequent checkpointing. Checkpointing is known to be extremely expensive, often dominating performance, and reducing the frequency of checkpointing can account for dramatic performance improvement. Now, suppose that computation of a task produces incremental results, which also provide a starting point at the time of restart. If these incremental (intermediate) results of a computation are multicast to those tasks that depend on it, then in the event of a task failure and restart,

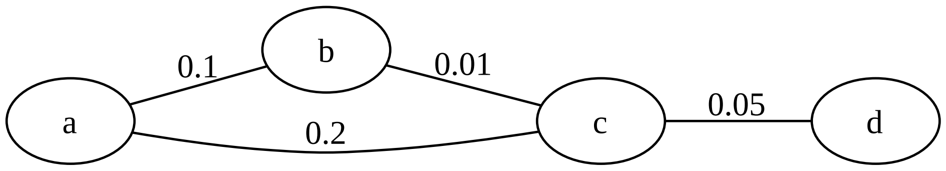

Figure 4.2. An example of a simple substitution graph involving four tasks. Numbers above edges represent weights on substitutions, that is, errors incurred due to these substitutions.

the task can receive these intermediate results from its dependencies without requiring any checkpointing (except, perhaps, that required to start up the task at all). As a result, checkpointing can perhaps be considerably less frequent. While such "hotstarts" are not yet a part of our implementation, they could provide an important part of the proposed architecture without very much added effort.

The dependency graph is one central data structure that allows us to implement decentralized fault tolerance and, to some degree, fault obliviousness. The central structure that targets specifically fault obliviousness (to the extent it is possible) is the substitution graph, which we discuss presently.

4.2 Substitution Graph

A substitution graph specifies, for node *i*, a collection of nodes that generate data that can substitute for the data generated by *i*. As an example, consider a pair of states in value iteration that transition to each other with high probability for some actions, and suppose that the discount factor is not too low. The values at these two states are natural substitutes, since each of these can be reached from the other in just a single step. While it is not necessary that substitution relationships are reciprocal, it is quite natural that they are, and we therefore assume that the substitution graph is undirected. Note that while it seems also natural that these relationships are transitive, they certainly need not be. The weights on substitution links represent errors, and it is only meaningful to include edges between tasks that can substitute at some relatively small loss; hence, transitivity may easily fail because errors, individually small enough, exceed the threshold once added together. An example of a substitution graph is shown in Figure 4.2.

The substitution graph mediates fault obliviousness as follows. Suppose that a processor (and a corresponding task) fails, but there is a good substitute for this task (in terms of generated data). The tasks that depend on it may then use the best substitute, rather than requiring the failed task to be restarted (which may happen anyway, but we no longer need to worry about any computational expense associated with task restart).

In many domains, closeness of two tasks would depend on who needs the resulting data. If we implement the substitution graph in a decentralized fashion, allowing this is a trivial generalization, since each node would simply store internally its private set of substitution weights for its dependents.

4.3 Trading Off Replication and Substitution

The main operation of the DAS architecture would trade off replication and substitution by attempting to substitute for tasks as long as it is efficacious to do so. Specifically, suppose that we set an upper bound on the error tolerance of the job, say at ε. Let us first consider a simplification where, when failures occur, we take a myopic point of view that no further error conditions will occur before the conclusion of the entire job. We now introduce some formal notation that will allow us to pose the tradeoff we wish to address as a *mixed integer program*. First, suppose that all errors are additive (e.g., using the l_1 norm). Let w_{ijk} be the error incurred when task j is substituted for task i as input to a task k that depends on i. Let I be the set of tasks that have failed and D_i denote the set of tasks that depend on a task $i \in I$. We denote by v_i a decision variable that is 1 if task i is to be substituted for (rather than replicated), and let z_{ijk} denote a decision to substitute task j for i for a dependent task k (thereby replacing k's dependence on i with j). We then obtain the following mixed integer program (MIP):

$$
\begin{aligned}
&\max_{v,z} \sum_{i \in I} v_i \quad \text{subject to:} && && (4.3.1)\\
&\text{Error budget:} && \sum_{i \in I} \sum_{j \notin I} \sum_{k \in D_i} w_{ijk} z_{ijk} \leq \varepsilon, && (4.3.2)\\
&\text{Single substitute for } i, j\text{:} && \sum_{j \notin I} z_{ijk} = v_i \quad \forall i \in I, k \in D_i, && (4.3.3)\\
&\text{No failed tasks:} && z_{ijk} = 0 \quad \forall i, j \in I, k \in D_i, && (4.3.4)\\
&\text{Binary variables:} && v_i, z_{ijk} \in \{0, 1\}. && (4.3.5)
\end{aligned}
$$

Note that Constraint (4.3.4) is actually unnecessary to specify in practice, since we can simply ignore the corresponding entries of z in implementing the optimal policy; Constraint (4.3.3) already ensures that there is exactly one substitute from only the functioning tasks for any task dependency pair if and only if the corresponding $v_i = 1$.

While the number of variables and constraints is polynomial, solving this integer program is likely infeasible in many realistic cases, and simplifications may need to be made in order to do so in real time. Denote by N_i the number of tasks that depend on i (and that will be using the substitute). One simplification is to restrict attention to w_{ij} without reference to k (or maximal over all k), then focus only on best substitutes for any failed i, letting $w_i = N_i \min_{j \notin I} w_{ij}$, and allow only a single substitute for any task. This results in the following simpler MIP:

$$\max_v \sum_{i \in I} v_i \quad \text{subject to:} \tag{4.3.6}$$

$$\text{error budget:} \qquad \sum_{i \in I} w_i v_i \le \varepsilon, \tag{4.3.7}$$

$$\text{binary variables:} \qquad v_i \in \{0,1\}. \tag{4.3.8}$$

Observe that the resulting MIP represents a classic knapsack problem, which, while NP-hard, can be approximated by a greedy algorithm which adds tasks to be substituted in increasing order of w_i, until the error budget is saturated. This greedy heuristic would be fast enough to be run in real time and is, in fact, what we have implemented in the simulation below.

4.4 Allocating Tasks to Processors

For the discussion above, we have often implicitly assumed that a single task is allocated to a processor (although in most cases we have not lost any generality in our discussion). This is also a very explicit assumption in the simulations below. Realistically, of course, multiple tasks will be allocated to a single processor, and the question of interest in our framework becomes two-fold: (a) how to allocate tasks to processors so that there is minimal interdependence between processors and (b) how to substitute in a way that preserves this initially low interdependence. To begin, let us assume that no substitution is allowed. The problem is then an instance of *graph partitioning* [6]. In a graph partitioning problem, the goal is, informally, to partition a set of nodes in a graph such that the weighted sum of nodes in each partition is bounded by some positive integer K and the weighted sum of edges between partitions is bounded by another positive integer J. This problem is known to be NP-complete, although algorithms to solve it exist, including in the Zoltan library [4]. In our case, all weights would be 1. To incorporate information about substitutes, we can superimpose the two graphs, but choose weights on substitute edges to be lower than those on dependency edges (to reflect that we may not need to substitute at all). Additionally, weights may decay as the substitution weights w_{ijk} increase. We can also add the graph partitioning constraint on substitutions when tasks fail, requiring that, upon substitution, no more than J dependencies cross processor boundaries. Formally, letting $p_{jk} = 1$ if and only if tasks j and k run on different processors, we constrain that

$$\sum_{j \notin I} \sum_{k \in D_i} p_{jk} z_{ijk} \le J \quad \forall i \in I. \tag{4.4.1}$$

in the first MIP. In the second, simplified, MIP, let $p_{ik} = 1$ if and only if the best substitutes for i and task k lie on different processors. We then constrain that

$$\sum_{k \in D_i} p_{ik} v_i \le J \quad \forall i \in I. \tag{4.4.2}$$

Note, however, that in this case the simple greedy algorithm will no longer apply to solve this problem and alternative, custom heuristics or algorithms are required to approximate it in real time.

4.5 Running Tasks with a Limited Number of Processors

One important subproblem when resources are severely constrained is to determine whether it is possible to run a subset of tasks on available processors, given a specified dependency graph. Here, imagine a case where there are not enough processors to run the entire job. In principle, if all tasks are interdependent, it may be impossible to make any progress until all tasks can be allocated concurrently. However, if dependencies are sparse, it may well still be feasible to make progress on the problem by running a subset of tasks that is independent of any others. It turns out that this problem can be solved in polynomial time by the following algorithm (assuming here again that each task is mapped to a single processor):

```
1. Let $T$ be the set of all tasks, $K$ the number of available processors
2. For each task $i \in T$,
     • Let $D = D_i$
     • For each $j \in D$,
         – set $D \leftarrow D \cup D_j$
     • until $D_j \subset D \; \forall j \in D$
     • If $|D| \leq K$, return $D$
3. return 0
```

The running time of this algorithm is $O(n^3)$ (where n is the number of tasks), and it will either return a set of tasks that has no internal dependencies, D, such that $|D| \leq K$, or 0 if such a set cannot be constructed. In fact, it runs in time $O(n^2)$ if the number of dependencies between tasks is bounded. Since we follow the dependency links from all possible starting points (all tasks), this algorithm is complete. The problem is that even though it runs in quadratic time if the maximum number of dependencies is bounded by a (small) constant, this is still too slow to do in real time in an exascale computing system. One solution is to only perform the procedure until we run out of search time (perhaps on a random permutation of tasks) in the outer loop, and only for a small number of iterations at the inner loop. There may also be a faster algorithm (on average) than the simple one described above, that would allow this problem to be solved very fast exactly in most cases.

4.6 Empirical Substitution Weights

The main question relating to the substitution graph is how to obtain the substitution weights. One possibility is that they are given (as upper bounds, perhaps, or some crude approximation) by the job developer himself, who knows something about the relationship between tasks. That may be reasonable in some scenarios, but would usually place a high burden upon the programmer. As an alternative, we may consider deriving them empirically. One general paradigm is to assume that the data generated by tasks (over time) constitute a sequence of real vectors. Given such sequences for two tasks that are meant to substitute for each other, we can create an empirical measure of the substitution weight as the actual error between the generated data streams. For example, if the error between the entire data streams is important, we would use something like Hausdorff distance between the two sequences, while if only the latest updates are significant, it would suffice to measure, say, l_1 distance between the latest data generated (as is the case in our grid world example below). We may still wish for the programmer to specify the actual substitution weights, as well as initial weights, but neither is strictly speaking necessary: We can measure weights between pairs of tasks, and add a substitution edge if the empirical weight is below some threshold (or, instead, place a threshold on the number of edges, and only add those with the lowest empirical weight). If the programmer does not specify initial weights, we can take them to be infinite by default, forcing replication until sufficient data about tasks is obtained to make substitutions useful.

4.7 Simulation and Results

We use simulations to provide a limited evaluation of our DAS architecture, implemented in Java. These are centered on the application to distributed asynchronous implementation of value iteration in the domain of a "grid world", which we now describe.

4.7.1 The Grid World

The grid world is a simple geographical representation of an agent walking in two dimensions. In our even simpler representation, an agent is allowed at most four actions in each cell: left, right, up, and down (corresponding essentially to the directions of a walk). The catch is that a walk to the right does not necessarily result in the agent ending up in the cell immediately to the right. Rather, he moves in the direction of his action with some probability (0.8 in our implementation), while the remaining probability is divided evenly among all the remaining physically adjacent cells (as well as current cell). The rewards of states are generated independently following a distribution with $\Pr\{r(s) \leq r\} = r^{1/3}$. Figure 4.3 shows an example 3×3 grid world.

In our implementation we set the discount rate δ to be 0.95, and let the number of states vary (although always maintained as a square grid). To prevent computation from proceeding indefinitely, we also set a stopping criterion to be convergence within 0.001.

Based on the grid world model, we generate the dependency graph by adding links in both directions between any two neighboring cells. Since neighboring cells also provide good substitutes for each other, we add links between all neighbor cells into the substitution graph. Initial (or default) weights are generated as upper bounds on the difference between final state valuations. These differences can be bounded by observing that, if s and s' are neighbors, then

$$V(s) \geq \delta P^a_{ss'} V(s') \tag{4.7.1}$$

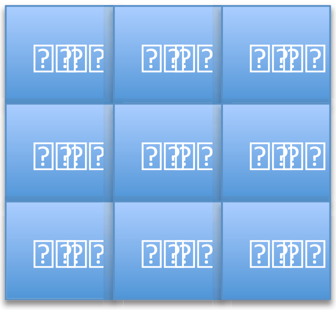

Figure 4.3. Example of a 3×3 grid world with the corresponding state reward structure.

or

$$V(s') \quad V(s) \leq V(s')(1 \quad \delta P^a_{ss'}) \tag{4.7.2}$$

for any action a, and, simultaneously,

$$V(s') \geq \delta P^{a'}_{s's} V(s) \tag{4.7.3}$$

or

$$V(s) \quad V(s') \leq V(s)(1 \quad \delta P^a_{s's}) \tag{4.7.4}$$

for any a'. Taking a and a' to maximize the right-hand side in Equations (4.7.1) and (4.7.3), we get the tightest bounds. This is the case when the action in the grid world is toward s' and s respectively, which achieves the fixed transition probability p that an agent ends up in the direction he tried to follow. Further, if δ is the discount factor, then $V(s) \leq 1/(1 \quad \delta)$ for any state s. Combining, we get

$$|V(s) \quad V(s')| \leq \frac{1 \quad \delta p}{1 \quad \delta}. \tag{4.7.5}$$

Note that if p is large, the difference between values of neighbor cells is tightly bounded, while with a small p, this bound is loose. In any case, this provides either the actual or initial substitution weights in our simulations (actual if we turn off empirical tuning of substitution weights, and initial if we turn it on).

4.7.2 Simulation Setup

In order to perform a first-order analysis of the proposed architecture, we developed simulation software that generates sample grid worlds and performs asynchronous distributed value iteration, with the architecture governing how tasks are allocated computing time on a simulated cluster. Time in the simulator is discrete, and we run it for 100 iterations (time units). Given the high discount rate, this ensures that tasks rarely complete in the allotted time, even if no processor failures occur. We assume, furthermore, that processors fail independently with probability p_b,

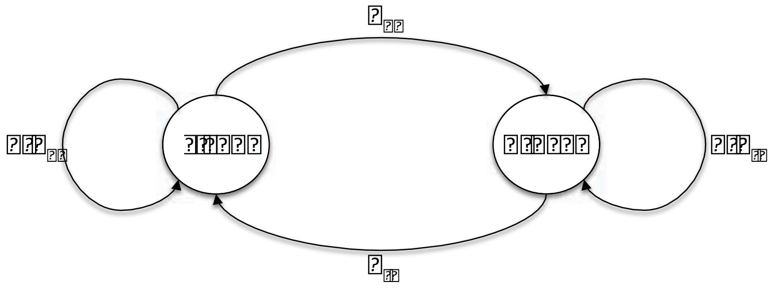

Figure 4.4. The 2-state Markov chain model of the processor failure and repair process.

and a broken processor is fixed with probability p_f. Figure 4.4 shows the resulting Markov chain process that models transitions between a fixed and broken state for each processor.

Lemma 4.7.1 [Steady State] Let π_f and π_b be the steady state probabilities of being in a fixed and broken states respectively. Then

$$\frac{\pi_f}{\pi_b} = \frac{p_f}{p_b}. \tag{4.7.6}$$

Proof. That there is a steady state is trivial here, and by the definition of steady state, $\pi = P\pi$, where P is the matrix of state-to-state transition probabilities. Thus, $\pi_f = (1 \quad p_b)\pi_f + p_f \pi_f$ and $\pi_b = (1 \quad p_f)\pi_b + p_b \pi_f$. Solving this system of two equations gives the result. □

As a consequence of this lemma, we can observe that the main parameter that governs the fraction of time each processor spends in a fixed and broken state (and, thus, the expected number of fixed and broken processors) is $\alpha = p_f/p_b$. For example, setting $\alpha = 1$ ensures that 1/2 of all processors are functional on average in the steady state. In the simulation, we let the number of processors be twice the number of tasks, and set $\alpha = 1$, thereby focusing primarily on the variance due to a specified failure probability (since, on average, there are enough processors to run all the tasks, but often in reality there will not be). Furthermore, we initialize the processors to be in a broken and fixed state precisely according to the corresponding steady state fractions. Our choice that the steady state fraction of available processors matches exactly the number of tasks that need solving may seem idiosyncratic, since it is easy to ensure that we have enough tasks such that they are not prone to processor failures (and, thus, at most we only require replication, but not substitution of tasks). Note, however, that we in general wish to utilize all the available resources: in a way, if we are, say, discretizing our problem more coarsely due to system considerations, we are in effect wasting available system capacity, which is certainly undesirable. Furthermore, such simplifications introduce error into our problem, which we can potentially avoid by fully utilizing available processing capacity. Thus, we would actually expect that the system capacity is fully used (which in our case means that all processors that are available in expectation are used to run tasks), and our assumption is not so outlandish.

4.7.3 Performance Measures

In our simulations we evaluate two extreme policies: the first allows no substitutions at all, whereas the second allows unlimited substitution. Thus, neither policy actually involves solving the optimization problems described above, and our results are in a sense unfavorable, since it is quite likely that in fact a mix of replication and substitution is necessary for good performance.

The first performance measure is the probability that the tasks make any progress in a given iteration (note that since all the tasks in the grid world example are initially interdependent, either all or none can make progress, depending on processor availability). The second performance measure is the l_1 error (relative to failure-free runs) of the computed state values.

4.7.4 Simulation Software

The key to our simulation software is a collection of Java interfaces which essentially describe its high-level operation:

- *World*
- *Task*
- *Cluster*

The *World* interface features the following methods, which in essence describe an MDP:

- *getReward(state):* returns the reward for a specified state
- *getAvailableActions(state):* returns the set of actions available in the specified state
- *getReachableStates(state):* returns the set of states that can be reached directly from current state with non-zero probability
- *getTransitionProbability(curState, nextState, thisAction):* returns the probability of transitioning from *curState* to *nextState* if action *thisAction* is taken

The *GridWorld* class implements the *World* interface, specializing it to the two-dimensional grid world described above.

The *Task* interface is the central interface to tap the dependency and substitution graphs:[1]

[1]We have implemented it in simulation not entirely the way we described such decentralized implementation above, primarily to make implementation slightly simpler (since we merely want a simulation tool, not the actual architecture); it is natural to implement the decentralized substitution graph as above, however.

- *getData(String msg):* simulates a message passing interface for querying the needed data; in practice we wish to also have a multicast interface, which could be simulated by a sendData() method, though we did not include that in our simulated implementation
- *getDependencies():* returns all tasks that this task depends on
- *getDependOnMe():* returns all tasks that depend on this task
- *getSubstitutes():* returns all tasks that substitute for this task (together with corresponding substitution weight, via a class *TaskWeight* that stores two substitute tasks and the substitution weight
- *getBestSubstitute(failedTasks):* returns the best (lowest weight) substitute to this task that is not one of the specified failed tasks
- *replaceDependence(toReplace, replaceWith):* replaces a (failed) dependence *toReplace* with *replaceWith*; this allows implementation of the substitution mechanism described above
- *updateState():* simulated interface to allow computation to occur in this task
- *completedTask():* returns true if the task has completed its computation

The *Task* interface is implemented by a combination of *AbstractTask*, which implements the dependency and substitution graph (and would be a part of the architectural/OS platform), and *GridWorldTask*, which implements the methods as relevant to the specifics of the grid world.

Finally, *Cluster* interface describes the methods associated with a cluster (or any multiprocessor/multicore system):

- *runCluster(numIterations):* run the computation on the cluster for a specified number of iterations (100 in our implementation). Our implementation simulates asynchrony by running each task in a given iteration with a specified probability (0.9 in our implementation).
- *addTask(task):* adds the specified task to the task/job queue
- *removeTask(task):* removes the specified task from the task/job queue (if it is running, kill it)
- *getActiveTasks():* returns the set of tasks that have not yet completed or been removed from the queue

4.7.5 Simulation Results

We ran simulations with two problem instances, one with 100 tasks, the other with 2500. While both are a far cry from the exascale environment that we target, they allow us to focus on the primary issues that concern us: resilience in the face of frequent failures and scalability, at least

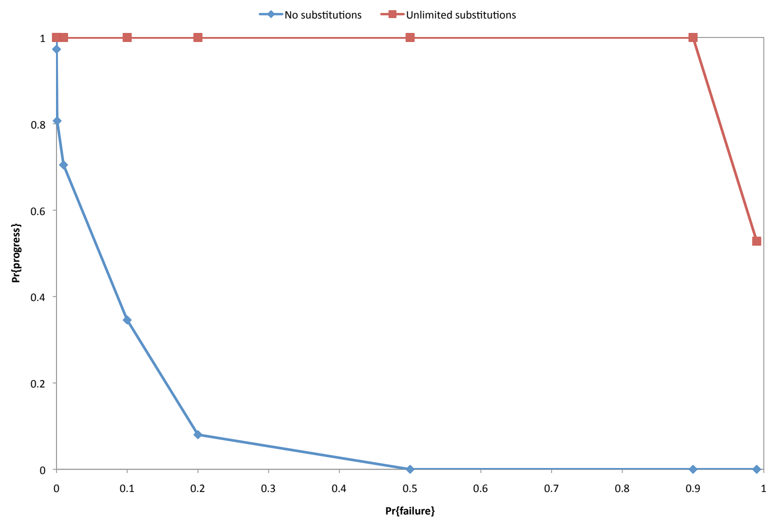

Figure 4.5. Probability that tasks are making progress on a 10×10 grid world as a function of failure probability (keeping $\alpha = 1$ fixed).

to a limited degree. The results we present are averaged over 150 random realizations of the grid world model. Our initial inquiry concerns the first performance measure: probability that any progress is made in a given iteration. Figures 4.5 and 4.6 plot the probability of making any progress on a 10×10 and 50×50 grid, comparing a case where no substitutions are allowed with one that allows unlimited substitutions. These figures demonstrate very strongly the resilience of the jobs due to the substitution framework: With arbitrary and unlimited substitutions, failure probability has to approach 1 before progress is even somewhat halted. By comparison, progress ratio drops dramatically with increasing failure probability when no substitutions are allowed.

Considering now the l_1 error measure, our results are somewhat mixed (see Figures 4.7 and 4.8). On a 10×10 grid, it appears that substitutions do introduce greater overall evaluation error. However, a 50×50 grid shows an unambiguous advantage to substitutions, so it seems that substitutions are more advantageous with greater scale, something of great relevance to us since we are motivated by exascale computing. In these figures, we also compare the case when substitution weights are the constant bounds that we derived above, or are empirically derived by directly comparing task data. Surprisingly, even though we do not impose a limit on substitution weights, empirically derived weights show a clear advantage, most likely because even though we allow unlimited substitutions, the *choice* of which tasks to substitute is driven by optimization, and finer-grained information about the resulting errors allows us to make better decisions.

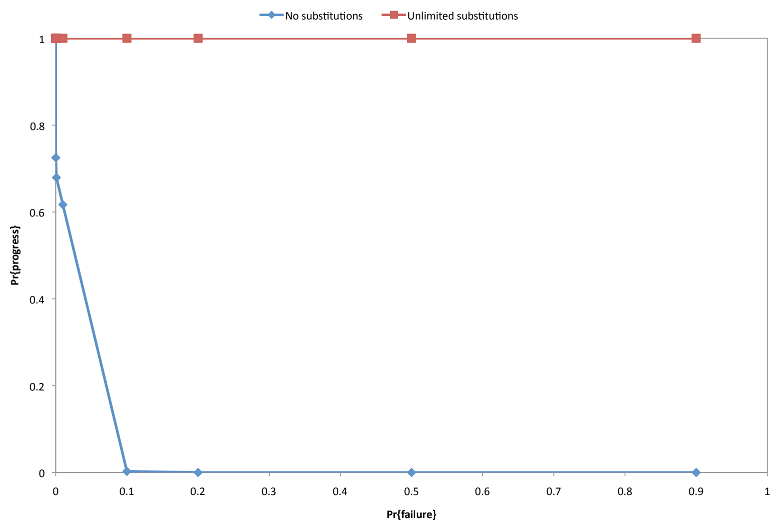

Figure 4.6. Probability that tasks are making progress on a 50×50 grid world as a function of failure probability (keeping $\alpha = 1$ fixed).

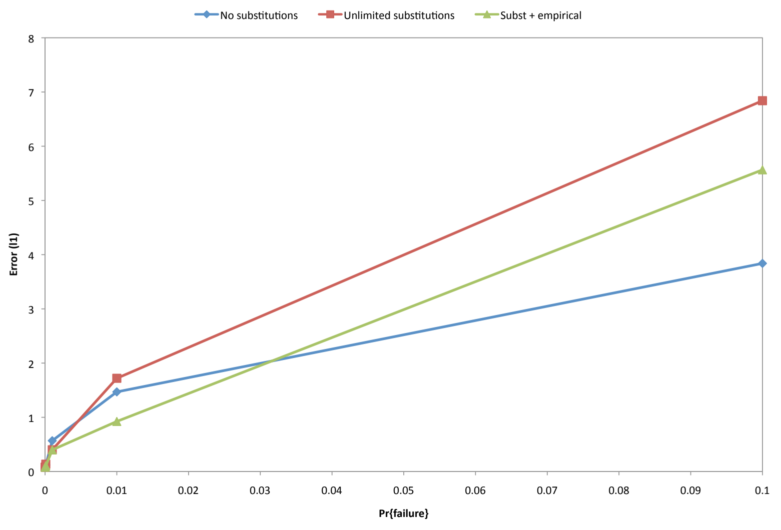

Figure 4.7. l_1 error (over all state values) on a 10×10 grid world as a function of failure probability (keeping $\alpha = 1$ fixed).

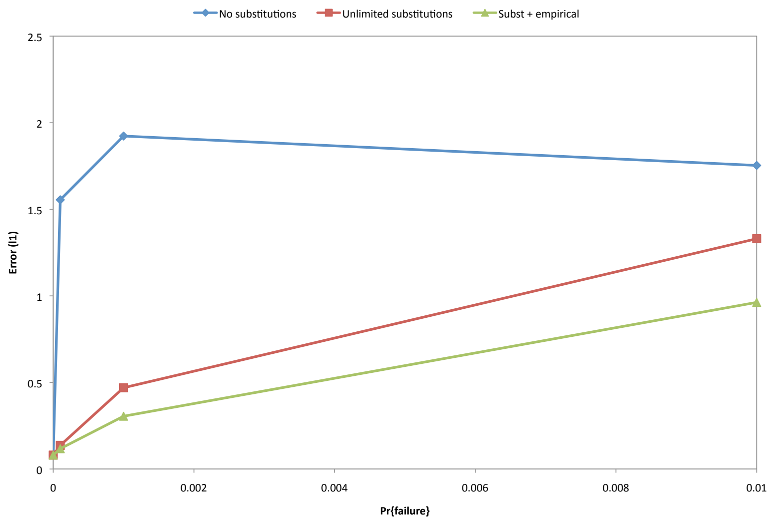

Figure 4.8. l_1 error (over all state values) on a 50×50 grid world as a function of failure probability (keeping $\alpha = 1$ fixed).

Chapter 5

Emulation Technology

Building on related work implementing the sandpile model as a emulated representation of a botnet, we developed a preliminary implementation of the Jacobi relaxation solver described in Section 3.2 in a large-scale emulation environment. We have not fully tested the Jacobi relaxation implementation or characterized its behavior quantitatively in this project. But we have successfully run the implementation at a scale of tens of thousands of virtual nodes, in a setup where hundreds of the virtual nodes typically fail, and have observed the expected characteristic patterns of fault-oblivious behavior.

5.1 Experiment Setup and Tools Used

The hardware in this experiment consisted of 128 Dell 1850 servers that were oversubscribed to boot 32,768 virtual Linux instances. Virtual LANs (known as VLANs), gateways, and routers were constructed to emulate the configuration of modern large-scale HPC platforms. The system software used in this exercise, called VMatic and Pushmon, was developed at Sandia [12].

5.1.1 VMatic

VMatic is a tool that assists with the creation of the emulated environment. It allows for the rapid provisioning and configuration of virtual machines by extending the OneSIS cluster management program. Via computational configuration of system properties such as networks, VLANs, or run level programs, VMatic makes it possible to instantiate a custom environment quickly and reliably.

5.1.2 Pushmon

Pushmon is a hierarchical program for monitoring the system- and application-generated data on virtual machines, physical hosts, and routers. The tool is designed to minimize system perturbation whenever possible, taking advantage of virtual machines' relationship to their physical host. Collections of data between virtual machines are passed through an out-of-band virtual block device layer that is shared between the hosts and all virtual machines, thus removing the need for traversing through the network subsystems.

5.2 Emulation Overview

By using emulation, we are providing a more realistic environment that allows for more data parallelism than what can be provided from simulation alone. Our approach involves the oversubscription of physical resources to allow the instantiation of true operating systems on top of virtualized hardware. For this particular experiment, the goal was to create a network environment that would be able to reach to largest possible scale given the amount of time and resources at our disposal.

Emulation can be used as a validation mechanism for testing the effectiveness of algorithms and proposed policies that are being considered. While simulation provides a useful insight into the functioning of these proposed changes, emulation can provide better fidelity by using the actual code and more realistic hardware that is used in practice. We have seen cases where the actual implementations of strategies have had consequences that could have only have been realized on an emulation of the actual network using the same system calls and operating system code that are used in production. In our computational experiments, modeled on the problem of heat transfer through a metallic plate, we have witnessed properties of the Jacobi relaxation algorithm that permit failures of computational resources to be interpreted as local abnormalities in the composition of the plate. These features are of interest because of the resilience to random abnormalities or failures in this system.

5.3 Jacobi Relaxation Experiment Deployed on the Emulator

In this experiment, we have implemented the behavioral properties of heat transfer through a metallic plate in software, as well as using it for a network architecture deployed on an HPC platform. A previous cellular automaton implementation of the sandpile model has been modified to reflect this heat transfer behavior. To test this networking model, we operate with a square lattice as the fixed domain of nodes for the entire experiment. This is not unlike current HPC applications that are executed in the MPI programming model where nodes are statically defined for each job execution.

In our experiment we emulate network communication defined by a set of rules that cannot be changed. A node may only directly communicate with an adjacent peer that is north, south, east, and west of the cell itself, and when the cell communicates with a peer it is allowed to only send one message to that peer. Cells are oblivious to the state of their peers and will send messages to peers regardless of the peer's actual existence. If the peer has died or fails to accept the message, the message will simply be lost and no re-transmission of the message will occur. The peer-to-peer interactions are in fact oblivious to the state of neighboring nodes in the lattice and are also oblivious to the overall nature or behavior of the system as a whole. Each individual cell behaves an independent automaton relying only on input from its neighbors to drive its behavior. It is important to note that nodes are real instances of the Linux operating system. When a network operation or any system call occurs, a full traversal through the operating system subsystem code takes place as if it were being executed on real hardware, the major difference being a slower code execution speed.

Failures in this model are handled in an unobservant manner. Having no dependent state is part of the network implementation, making the network capable of communication throughout the system in the presence of faults as long as a critical threshold of failed nodes is not exceeded. This is already an improvement on current programming models in which *single component failure* constitutes a failure of the entire system regardless of the criticality or significance of the data in question. That data loss may or may not affect the overall result of the computation, but the fact that a failure of any kind has occurred would traditionally be enough to stop the run from progressing.

Node placement in the overlay network creates the set of connections, and is extremely important because of the need to minimize the grouping of failures that are all within a close proximity region. Gaps in the overlay network change the behavior of network communication patterns, at best by changing the timing with which a certain result will occur, or at worst by preventing the result from occurring at all. This is not unlike how critical services are structured on the Internet, taking DNS as an example. Clone or slave DNS servers are replicated throughout different geographical locations, minimizing the amount of shared resources (such as power infrastructure or routers) that could create a failure on all redundant servers.

In the setup of the Jacobi relaxation implementation deployed on our emulated environment of virtual machines, peers are randomly distributed across the two-dimensional square lattice of the overlay network. The emulation of thousands of nodes is not only to represent a size closer to that of an exascale system but also to demonstrate the effect of physical node failures on the functionality of the overlay network. Since hundreds of virtual machines are sharing the same critical resources – i.e., the same physical host – a node failure can be likened to a power outage of an entire rack unit on an exascale machine. In the same manner that heat will propagate around a small hole, our nodes communicate around a small gap in the P2P lattice, giving us the ability to observe when and how critical thresholds that break down normal behavior occur.

References

[1] T. Archibald, K. McKinnon, and L. Thomas. Serial and parallel value iteration algorithms for discounted Markov decision processes. *European Journal of Operations Research*, 67:188–203, 1993.

[2] P. Bak, C. Tang, and K. Wiesenfeld. Self-organized criticality: An explanation of $1/f$ noise. *Phys. Rev. Lett.*, 59:381–384, 1987.

[3] J. Brandt, A. Gentile, J. Mayo, P. Pébay, D. Roe, D. Thompson, and M. Wong. Methodologies for advance warning of compute cluster problems via statistical analysis: A case study. In *Proc. Resilience 2009 Workshop, part of 18th ACM International Symposium on High Performance Distributed Computing*, Garching, Germany, June 2009.

[4] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. 21st IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, Mar. 2007.

[5] J. Daly. A model for predicting the optimum checkpoint interval for restart dumps. In P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, editors, *Computational Science – ICCS 2003*, volume 2660 of *Lecture Notes in Computer Science*, pages 3–12. Springer, 2003.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[7] S. Genaud and C. Rattanapoka. P2P-MPI: A peer-to-peer framework for robust execution of message passing parallel programs on grids. *Journal of Grid Computing*, 5:27–42, 2007.

[8] V. Gullapali and A. G. Barto. Convergence of indirect adaptive asynchronous value iteration algorithms. In *Neural Information Processing Systems*, pages 695–702, 1994.

[9] Kademlia. `http://en.wikipedia.org/wiki/Kademlia`.

[10] S. A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, 1993.

[11] J. R. Mayo, R. G. Minnich, D. W. Rudish, and R. C. Armstrong. Approaches for scalable modeling and emulation of cyber systems: LDRD final report. Sandia Report SAND2009-6068, 2009.

[12] R. Minnich and D. Rudish. Ten million and one penguins, or, lessons learned from booting millions of virtual machines on HPC systems. In *Proc. Workshop on System-level Virtualization for High Performance Computing in conjunction with EuroSys'10*, Paris, France, Apr. 2010.

[13] P. Porras, H. Saidi, and V. Yegneswaran. An analysis of Conficker's logic and rendezvous points. `http://mtc.sri.com/Conficker/`.

[14] Relaxation method. `http://en.wikipedia.org/wiki/Relaxation_method`.

[15] R. S. Sutton and A. G. Barto. *Reinforcement Learning*. The MIT Press, 1998.

[16] S. Wolfram. *A New Kind of Science*. Wolfram Media, 2002.

DISTRIBUTION:

1	MS	0899	Technical Library, 9536 (electronic copy)
1	MS	0123	D. Chavez, LDRD Office, 1011

Sandia National Laboratories