Improving Application Resilience through Probabilistic Task Replication

Tyler A. Simon and John Dorband
University of Maryland Baltimore County
Department of Computer Science and Electrical Engineering
{tsimo1, dorband}@umbc.edu

ABSTRACT

Maintaining performance in a faulty distributed computing environment is a major challenge in the design of future peta and exa-scale class systems. Better defining application resilience as a function of scale, is a key to developing reliable software systems and programming methodologies. This paper defines the resilience of a task as the survivability of that task (i.e., how well will it survive until it completes). Resilience varies with mean time to failure (MTTF) and inversely with runtime. We develop an approach for defining a resilience index(RI) for applications running on a system with a fixed MTTF. Our approach, inspired by radioactive decay, defines an application as a collection of tasks, which we model as particles with an exponential decay rate and therefore measurable half-life. We determine the probability of the number of task failures for an application using a poisson distribution over the interval of the task lifetime. Further we have developed a distributed runtime system, ARRIA, that measures both system reliability and application performance at runtime, which schedules and replicates tasks based on the probability of failure and expected runtime. We demonstrate that the resilience index can help to better define the tradeoffs for the designers of future systems and developers of parallel software. Thus, we propose a formulation of application resilience that results in a resilience index. We evaluate some initial and fundamental properties of the resilience index as they relate to application performance on high performance computing systems composed of many components, each with varying degrees of reliability.

Keywords

resilience; reliability; fault tolerance; scheduling; high performance computing

1. INTRODUCTION

Measuring resilience for applications on High Performance Computing(HPC) systems is becoming an important consideration in the design of peta and exascale class systems that are intended to run tasks on the order of hundreds of millions to billions of tasks with a mean time between failures (MTBF) on the order of hours. The challenges of exascale computing are becoming better defined but provide mostly qualitative tradeoffs between reliability, performance, power and monetary cost. Prior definitions of resilience can be summarized as maintaining performance of a system in the presence of component failures.

Extending the idea of fault tolerance in regard to applications is becoming increasingly important as the size and complexity of HPC systems grow toward the peta- and exascales. How can we evaluate whether or not one program is more resilient than another? Research in HPC for resilience has been focused primarily on quantitative methods for reliability, and qualitative methods for resilience. This paper intends to address the need for a standard metric for resilience by which the HPC community can better communicate and develop reliability requirements that include performance guarantees. Such a metric is also beneficial for developers who will designing applications today that will run on an exascale system in the 2018-2020 timeframe. This paper quantifies the tradeoffs between improving application performance and application reliability.

The best candidates for an efficient and resilient exascale application are those that are embarrassingly parallel, thus execute tasks independently through a bag of tasks or work pool paradigm. Recent work by Raicu has identified several classes of scientific application that fit a many task computing (MTC) model [34]. These types of applications can be considered "naturally fault tolerant" which have been demonstrated by Engelmann and Geist, through simulation, that for petascale class machines, both global optimization and finite element methods can be broken down into independent tasks [16]. For this paper we consider reliability costs associated with highly scalable applications and how to quantify their contribution to resilience.

HPC System reliability will be decreasing as we move to exascale, and checkpoint/restart will likely be so costly that efficiently executing large, long running jobs will be prohibitive [8]. If we were developing a million core application for a billion core system and would like to get the best performance, should we break it up into smaller pieces or shorter pieces? What is a reasonable processor size and running time for the application? For those designing runtime systems and application middleware for future exascale HPC

systems, it is imperative to know how well we can meet the resilience requirements while maintaining an acceptable level of performance. To this end we present the following contributions in this paper.

- Defining a resilience index as a task decay rate
- A description of the runtime system (ARRIA) that implements the RI
- Experimental analysis of the ARRIA system with system failures

2. RELATED WORK

2.1 Resilience

The Resiliency Challenge is, the ability of a system with a large number of components to continue operations in the presence of faults [21] [4] [13]. DeBartelaben et al. defines resilience as it relates to systems as "The ability of a system to keep applications running and maintain an acceptable level of service in the face of transient, intermittent and permanent faults [13]." This definition of Resilience has also been applied to applications [29] where the focus is to continue running within some performance threshold, during one or more system failures. The purpose of this paper is to present a methodology to better quantify this threshold for a more general class of applications. We will also focus on better defining levels of service, such as maintaining accuracy and maximizing performance.

It is important to distinguish between system and application failures. A system failure prevents one or more components, either hardware or software from performing its intended function [9]. This is also true for an application, where a fatal error either stops a component of an application from running or interrupts correct execution. The application mean time to fatal error (AMTTF) is measured as the expected runtime prior to a fatal or fail-stop error. The AMTTF measurement is also dependent on the failure rates of the components that it is running on, thus we focus on this metric when looking to increase the resilience for a running application. The author's agree with Chandler et al., in that an exascale system must be truly autonomic in nature, constantly aware of its status, and optimizing and adapting itself to rapidly changing conditions, including failures of its individual components [6].

There has been considerable research on increasing application based fault tolerance as it relates to resilience. Lee [31] and Raicu [35] [34] investigate the reliable execution of many independent tasks. This is a reasonable approach in that an application is broken down into fine grain tasks, and we can choose to replicate selected "mission critical" tasks with the potential of forming task groups. With this approach each thread in a group may be allocated to a different computational resource. This is a common execution model for bags of tasks that are run in large scale Grid computing environments [28]. As pointed out by Lee, a system of this type provides a graceful path of performance degradation during system component failure, but is not resilient because it does not "assure continued operation of the system when resources become available". Lee further suggests that what is required for true application resilience are "policy-based methods for controlling replication" [31]. In addition to this work on policy based replication, more recently, Jones and Daly point out the benefits of application monitoring frameworks for fault detection, they show that "near-instantaneous" fault detection is not necessarily required to improve overall application and system performance [29]. As an application runs we would like to know the potential effect of the the system MTBF on performance and how to adequately compensate the application. More recently, Jones et. al demonstrated that for high failure rates, (MTBF=300 minutes), the percentage of jobs that experience an interrupt is around 6.5%, and just above 1% for 5 times the MTBF [29]. Our previous work demonstrated the benefits of workload scheduling HPC workloads using a dynamic priority queue with multiple performance objectives [36]. For this paper we hope to extend this work to include resilience as a component of a scheduling policy couples with a dynamic or autonomic runtime system. Agathos et al., have also developed a task based runtime system and demonstrated a 25% overhead reduction when creating tasks instead of threads for nested *parallel for* loops using OpenMP [1] [2]. Additionally, The approach used by Das, which uses selective replication for independent tasks [40] is similar to our approach.

2.2 Faut Tolerant Frameworks

There are several fault tolerant frameworks that have been designed to provide application level fault tolerance through message logging, [15], mpich-V [23] [25] [11]. Fu and Stearley provide valuable insights into mining large system logs for failure analysis and prediction [20], [38]. Chen and Dongarra demonstrate that checkpoints can be circumvented entirely by calculating checksums prior to and after execution for certain matrix operations [7]. Further research in "proactive fault tolerance" aims at migrating running applications from nodes that are about to fail, combining both prediction and fault classification [32] [17].

2.3 Checkpoint/Restart

There is also considerable work on improving existing checkpoint/restart for large scale systems, using adaptive checkpoint restart [27]. Mootaz and Plank identify some of the problems with checkpoint/restart and rollback recovery for petascale systems [14]. Their findings suggest that "the data-parallel model of most compute-intensive applications will force the [petascale] system to be continuously checkpointing at the extreme." Mootaz also suggests the importance of transparent and automatic checkpointing. There are considerable benefits to having a standard metric for determining application resilience as a function of performance. For example, the message passing interface (MPI) is the most common parallel programming library and runtime system. Regarding fault tolerance in MPI, Snir et al. state that "MPI does not provide mechanisms for dealing with failures in the communication system" and that it is up to the implementor of the MPI standard to "reflect unrecoverable errors as exceptions [37], pg. 369 ". Several efforts are currently underway to support independent task failures in MPI [19]. CIFTS [22] from Argonne National Laboratory provides a programmer API to wrap MPI calls with fault tolerant semantics, as is the case in another fault tolerant MPI

Table 1: System Mean Time Between Failure, in hours, for large scale installations [13]

Year	Cores	SMTBF(h)	System	Location
2002	8192	40	ASCI White	LANL
2004	3016	9.7	Lemieux	PSC
2007	6656	351	Seaborg	NERSC
2002	8192	6.5	ASCI Q	LANL
2004	15000	1.2	Google	Google,CA.
2006	131072	147.8	BlueGene/L	LANL

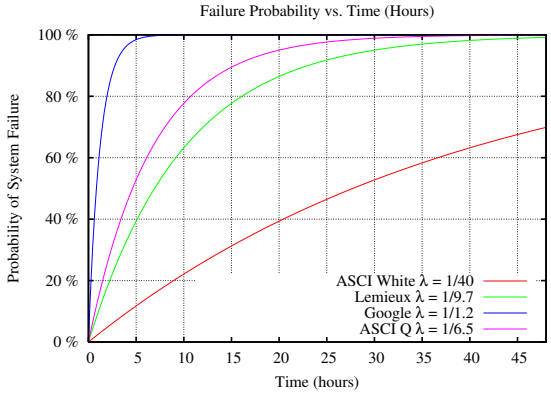

Figure 1: Exponential Failure Probabilities from [13]

implementation from the University of Indiana [26], specific fault tolerant syntax is introduced for the programmer.

3. RELIABILITY OVER TIME

3.1 The Standard Approach

If we have to restart a system once at the same time, every week then the system has a failure rate λ of (1/168)=0.0059, given as number of system failures over an interval of time, in this case hours $7 * 24 = 168$. The metric for the mean time to failure(MTTF) is then $1/\lambda$. For modeling faults on n equal systems we use the standard equation for the mean time to failure (MTTF) is given by the Equation 1 [41].

$$MTTF = \frac{1}{\lambda}\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) \quad (1)$$

Based on the measurements used from DeBartelaben et.al, and the measured values for system mean time between failure (SMTBF) in Table 1 from [13], Figure 1 shows an exponential failure distribution $F(t) = 1 - e^{-\lambda t}$, for selected measured failure rates from the table. In Figure 1 the horizontal axis shows time, and the vertical axis represents $F(t)$, or the probability of failure at time t for multiple values of λ.

3.2 Failures as a Poisson process

We can model failures as a poisson process such that the probability of failure occurring at any time is constant and failures are considered independent [41]. In a high performance computing system we can use the poisson probability distribution if we are interested in the number of failures for given time interval, for example, how many nodes we expect to lose over the next 48 hours. Generally speaking, if we know the MTTF for system X with size N, then the reliability at time t, $R(t)$, based on the exponential distribution is $R(t) = e^{-\lambda t}$ where $e^{-\lambda t}$ is the exponential factor in the poisson equation, $-\lambda t$ then becomes μ, as the *expected* number of failures for the entire system X. This is presented in Equation 2.

Obtaining the value μ for a particular application run on n nodes for time t, allows us to compute the probability of a subset of node failures, x, over the the duration of a job. In the case of running an application we are interested in the probability of losing n nodes in time t ($\mu = \lambda t$).

$$P(x) = \frac{e^{-\mu}\mu^x}{x!} \quad (2)$$

Additionally, we can model the poisson probability distribution $P(x)$ as a function of x and μ , this gives us the probability of having x failures over the lifetime of a process, which increases for longer or larger jobs. As an application with many tasks runs, knowing $P(x)$ becomes useful when number of recoverable failures are chosen by the application such that the application's completion is guaranteed, even with an increasing system failure rate. If we assume for an application running on n nodes, that the runtime is unknown, but we would like for it to be minimal, then this is the value we want to minimize, or maximize performance, $1/runtime$, as a function of μ and x. Additionally we want see the relationship between maximizing performance while minimize the probability for even a single failure of n nodes.

4. RESILIENCE AS TASK DECAY RATE

In our approach we define resilience as the ability of a system or application to resist errors, overcome errors and recover to some stable state until job completion. Associated with this definition is some guarantee of forward progress or performance in the presence of faults. Resilience of a task is the survivability of that task (i.e., how well will it survive until it completes), thus resilience varies with mean time to failure and inversely with runtime.

The resilience index should clearly distinguish, or quantify, differences in two extreme cases. In the worst case, if the overheads associated with a particular fault tolerance method, such as checkpoint/restart, takes longer than the MTTF for n components then the resilience index should reflect this as a low index value, showing there is little resilience. For systems with high MTTF we would expect that performance should not be hampered by application recovery as much and will therefore have a high resilience index.

We define the resilience index as the ratio $R = F/T$, where F is the MTTF and T is the runtime of a task. This reflects our reasoning that we should sustain a certain number of faults per unit time. If we assume that approximately half the tasks will fail in time F and that failures are independent of each other, F is analogous to a task failure half-life. The

number of tasks that will fail in time T is shown in equation 3 and the survival or resilience probability is shown in equation 4. If $R = 1$ then the probability of survival is approximately 0.5, if $R < 1$, the probability of survival is less then 0.5 and greater than 0.5 if $R > 1$. Note that if checkpointing or any other failure mitigation function is used, the time to perform the checkpoint must be included in T. Anything that decreases runtime or increases MTTF will improve resilience.

$$F_n(t) = n * (1 - 2^{-\frac{T}{F}}) \tag{3}$$

$$P(R) = 2^{-\frac{1}{R}} = 2^{-\frac{T}{MTTF}} \tag{4}$$

Regarding measurements for checkpoint overheads, John Daly provides a method for determining optimal checkpoint intervals [10], these are presented in Equation 6. We can use Daly's cost function of the total completion time of the application, the application walltime $Tw(\tau)$, in Equation 5, where τ is the compute interval between dumps. We identify overhead due resilience as $dumptime + reworktime + restarttime$, thus when these values sum to zero only the reciprocal of runtime accounts for performance. Figure 2 represents a plot from Equation 6 for various MTTI values on the X axis and δ values representing restart overheads in seconds. The colored values represent an optimal checkpoint interval and can be used as a lookup table in the overhead calculation for the resilience index.

$$\begin{aligned} Tw(\tau) \quad &= solvetime + dumptime + reworktime \\ &+ restarttime \end{aligned} \tag{5}$$

$$\tau_{opt} = \begin{cases} \sqrt{2\delta M} - \delta & \text{for } \delta < \frac{1}{2}M \\ M & \text{for } \delta \geq \frac{1}{2}M \end{cases} \tag{6}$$

Algorithm 1 illustrates the formulation of the resilience index as a function of checkpoint overhead, performance and MTTI (M). Figure 2 shows the resilience index after calculating an optimal checkpoint time δ, for a MTTF of 4 hours MTTI M for failure of a single task. The various lines in the figure are a result of multiple δ values. These results show the expected behavior from the proposed formulation and our future work will investigate calculating an accurate a resilience index on a per task basis.

5. TASK SCHEDULING AND RI

5.1 Multiobjective Task Scheduling

The task placement problem can be formulated as a multiple objective knapsack problem [39] [18] [30] [24]. This approach provides a simple and efficient algorithm for the simultaneous optimal solutions for mutually conflicting objectives, such as performance and reliability. In Huang et al. [24], Huang applies the multi-objective optimization problem to reliability and introduces the "Intelligent Interactive Multiple Objective Optimization" IIMOM model which uses an

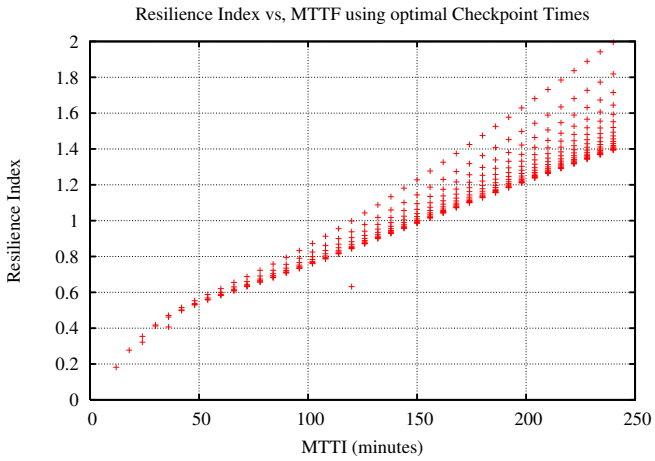

Figure 2: Resilience Index vs. MTTI and Checkpoint Overhead (δ)

Algorithm 1 TASKRESILIENCE(M, t)

Require: $M > 0$, $minimumfailures$
Ensure: The value of $R(M, t)$.

```
for i = δ to M do
    for j = δ to i * 0.5 do
        window = τ_opt(i, j)
        P(M, t) = 1 − e^{−λt}
        return Resilience(i, window, t, P(M, t))
    end for
end for
```

Artificial Neural Network to determine optimal model parameter vectors, which are used as input, and designer preferences over representative samples along the Pareto frontier as output. Similarly, we formulate the task scheduling problem into a multiple objective optimization problem. The standard formulation is presented in Equation 7 [12]. Through multiple objective optimization, a variety of solutions may be viable, typically these form a Pareto front, which provides a reasonable measure. In this paper we address the problem of finding the best balance of the conflicting objectives, maximize performance with the application resilience constraint. Our approach places tasks in a 0-1 knapsack shown in Equation 7.

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} R_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq N, \quad x \in 0, 1 \end{aligned} \tag{7}$$

Where n represents total processors, $R(i)$ the resilience Index and x_i are the cost values, in processors or tasks, and N is the capacity of the system.

One of the goals of this work is in searching the decision space to solve the above equations for maximum resilience. In our previous work we developed a hybrid scheduling algorithm for Many Task Computing workloads, which we

demonstrated a reduction of workload execution time using a dynamic priority queue [33]. here we address task scheduling for resilience using our proposed resilience index. As stated earlier typically evolutionary algorithms are used in solving multi-objective optimization problems. These algorithms can be effective in a dynamic distributed runtime system considering that the problem of scheduling tasks concurrently can be very effective [3] [5]. The task placement problem can be formulated as a multiple objective knapsack problem [39] [18] [30]. This approach provides a simple and efficient algorithm for the simultaneous optimal solutions for mutually conflicting objectives, such as performance and reliability. Further, our approach is novel in that we are extending these models to a massively parallel runtime system. The optimizations are transparent to the user, although a user may influence the importance of an objective, such as running an application in "highly resilient" mode. In this case a user may expect poorer performance but increased reliability of their computations. In section 6 we provide a brief overview of the ARRIA runtime system.

6. ARRIA RUNTIME SYSTEM

6.1 ARRIA Architecture

ARRIA stands for Adaptive Runtime for Resilient Applications. ARRIA is is a parallel runtime system that creates tasks and evaluates the resilience index for applications and consists of about 7,000 lines of C code. By gathering and using runtime performance metrics to both scale and manage user tasks as well as compute resources, ARRIA can vary the utilization of computational resources and for some applications increased scalability, performance and reliability. Through the use of a client API, ARRIA maps tasks onto compute resources using a decentralized, master-worker paradigm, this design is shown in Figure 3. We choose this approach because it is easily fault tolerant when tasks are independent and also scales well once a problem has been broken down into independent subproblems. ARRIA is ideal for executing loops iterations on independent data having the following form:

```
for i=1 to 100
do
A[i]=search(key,B[i])
done
```

6.2 ARRIA Components

The ARRIA system is composed of three fundamental and separate processes, each running as a concurrently executing thread and communicating over sockets using a simple messaging protocol. The communicating processes are a manager, client, and worker presented in Figure 4 and their capabilities are discussed below.

6.3 Client

The Client is an application interface that sends tasks the the manager via a ServiceRequest() routine. The input from the client, which is typically run in a traditional cluster environment is sent to the manager for use by the HPC system via SubmitData(). The client then awaits a response from the manager with the complete results through ListenforResponse(). Currently, the entire workflow latency, from Client

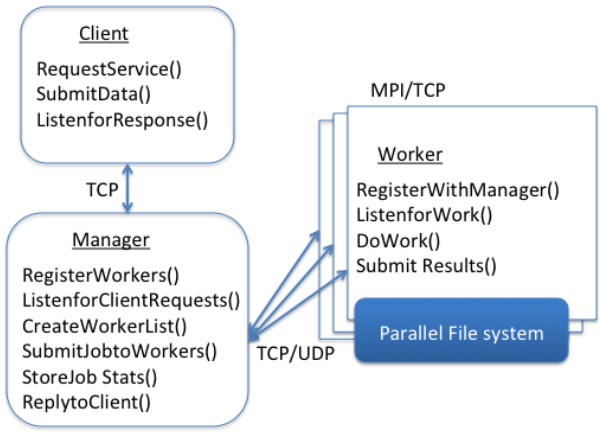

Figure 4: ARRIA Process Overview

to Manager, Manager to Workers, Workers to Manager and Manager to Client is on the order of tens of milliseconds when no task execution time is measured.

6.4 Manager

When the Manager is started it first receives a registration message from any participating Worker process from RegisterWorkers(), these can be within the local cluster or on remote machines. The only requirements for either a Client or Worker is the ability to establish a socket connection to the manager. The Manager the Listens for a ClientRequest() which it then serves with a list of services for the Client to populate it's interface listing. Once the Client has entered its parameters and data location the message is sent to the Manager, who compiles a list of capable Workers via CreateWorkerList(). With this information the manager selects a subset of workers to perform the execution of the client application. The manager also determines which tasks are to be replicated. The Manager then logs each job in a local database to be used for future reference and when a response is received form the work pool the manager checks for correctness and sends the results on to the Client.

6.5 Workers

Each Worker has a *Characteristics* structure, shown in Listing 1, which is sent as a status message to the Manager, which outlines its computational capabilities, such as load average, number of cores, core frequency, memory in use, memory available, bandwidth to local storage, bandwidth to remote storage and whether or not a GPGPU is present. The Worker then polls a work queue and if it is listed for job execution receives a message from ListenforWork() then executes the job and submits the results to the manager. The worker then updates an Experience structure, shown in Listing 1 with new information that can be used by the manager for use scheduling decisions regarding client task placement and replication.

6.6 Communication

All ARRIA communication is handled between processes via UDP for all system messages. Other communication mechanisms, such as MPI may be used between tasks, but tasks requiring message passing will be bundled together

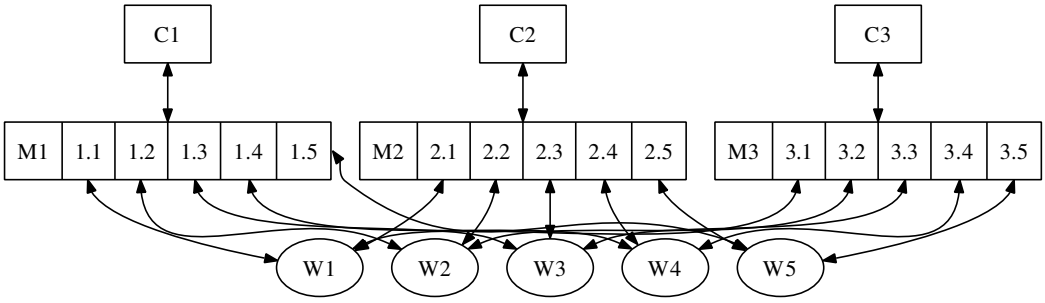

Figure 3: ARRIA Client(C), Manager(M) and Worker(W) communication

```
typedef struct WORKER_CHARACTERISTICS
{
    int nprocs_per_node;
    int cpu_frequency;
    float memory_available;
    float total_memory_capacity;
    int gpu;
} WORKER_CHARACTERISTICS;

typedef struct WORKER_EXPERIENCE
{
    float connection_throughput_local;
    float connection_throughput_remote;
    float memory_available;
    double loadavg;
    double numa_hits;
    int uptime;
    float MTTF;
} WORKER_EXPERIENCE;
```

Listing 1: Worker Characteristics and Experience Shared Structure

```
typedef struct ARRIA_HEADER
{
    unsigned char Type;
    unsigned char format;
    unsigned char algorithm;
    int GenericA;
    int GenericB;
    int GenericC;
    int SequenceID;
    int buffLength;
    char *buffer;
} ARRIA_HEADER;
```

Listing 2: ARRIA Shared Message Header

and treated as a single ARRIA task. All ARRIA processes communicate with a custom common message header that each process reads to determine the routing of messages, this header is shown in Listing 2. The key to ARRIA's simplicity is that messages are routed based on the *Type* field. This informs each process thread what to expect in the remaining fields. This field contains predefined numerical values, which can change and tell the reading process how to handle the message. The values for the *Generic* fields depend on the algorithm being executed and the message type. The *algorithm* filed can be any predefined function to be executed by a Worker, these include FFT's, sorting and other functions that could be done through the use of RPC's, where the requirement is valid input.

6.7 ARRIA Performance

One of the objectives in designing ARRIA is to overcome individual node level and task level failures. Only some tasks need to be replicated. Figure 5 shows the time taken to schedule and execute a 100 task, 100 second job on an 8 node, dual-quad core Intel cluster. The "Ideal" line, is the ideal runtime where each task takes 1 second to run. There are error bars present but show very little variation.

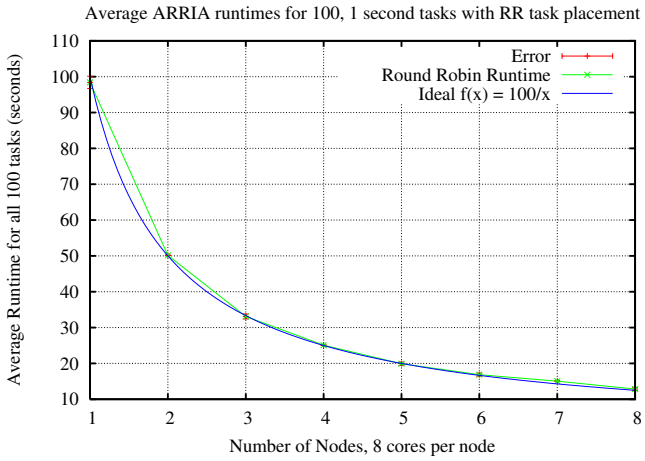

Figure 5: ARRIA Round Robin task placement with Failures

7. CONCLUSIONS AND FUTURE WORK

The purpose of this paper was to introduce the idea of a resilience index, define its properties and present a custom runtime system that takes advantage of the resilience index with a focus of relieving the user from explicit checkpoint/restart. We have presented an algorithm that enables us to determine the expected resilience of jobs based on their size, number of tasks, and their duration. Our approach relies on the concept of exponential decay. We evaluate our formulation of a resilience index that incorporates optimal checkpoint intervals and developed an API to generate tasks for users. Additionally we proposed the design of a dynamic runtime system, ARRIA, that can act as a resource manager and task generator from user applications. The ARRIA system has been tested with a variety of computationally intensive application utilizing more traditional HPC paradigms such as MPI, OpenMP and Hadoop's Map Reduce. We intend on moving the ARRIA framework forward to utilize high performance distributed shared memory programming paradigms such as OpemSHMEM and UPC. We also plan to incorporate custom graph analytics capabilities, particularly in large, federated and cloud data sources into the HPC framework as well as offloading user vectorized (SIMD) work to GPGPU's.

Acknowledgment

The work was supported by an NSF IUCRC grant and the Laboratory for Physical Sciences. The authors thank Milton Halem, David Mountain and John Daly for their comments and insights which led to this work.

8. REFERENCES

[1] S. Agathos, P. Hadjidoukas, and V. Dimakopoulos. Design and implementation of openmp tasks in the ompi compiler. In *Informatics (PCI), 2011 15th Panhellenic Conference on*, pages 265 –269, 30 2011-oct. 2 2011.

[2] S. N. Agathos, P. E. Hadjidoukas, and V. V. Dimakopoulos. Task-based execution of nested openmp loops. In *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 210–222, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] A. Benoit, L. Marchal, J.-F. Pineau, Y. Robert, and F. Vivien. Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. *Computers, IEEE Transactions on*, 59(2):202 –217, feb. 2010.

[4] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor, 2008.

[5] R. Bertin, A. Legrand, and C. Touati. Toward a fully decentralized algorithm for multiple bag-of-tasks application scheduling on grids. In *Grid Computing, 2008 9th IEEE/ACM International Conference on*, pages 118 –125, 29 2008-oct. 1 2008.

[6] C. F. Chandler, C. Leangsuksun, and N. DeBardeleben. Towards resilient high performance applications through real time reliability metric generation and autonomous failure correction. In *Proceedings of the 2009 workshop on Resiliency in high performance (Resilience '09)*, pages 1–6, Munich, Germany, June 2009.

[7] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Trans. Parallel Distrib. Syst.*, 19:1628–1641, December 2008.

[8] Z. Chen and J. Dongarra. Highly scalable self-healing algorithms for high performance scientific computing. *IEEE Trans. Comput.*, 58(11):1512–1524, Nov. 2009.

[9] J. Daly, L. Pritchett, and S. Michalak. Application mttfe vs. platform mtbf: A fresh perspective on system reliability and application throughput for computations at scale. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, pages 795 –800, may 2008.

[10] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, Feb. 2006.

[11] T. Davies and Z. Chen. Fault tolerant linear algebra: Recovering from fail-stop failures without checkpointing. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –4, april 2010.

[12] K. Deb and D. Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[13] N. DeBarteleben, J. Laros, J. Daly, S. Scott, C. Engelmann, and W. Harrod. High-end computing resilience: Analysis of issues facing the hec community and path forward for research and development, Jan 2010.

[14] E. N. Elnozahy and J. S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1:97–108, 2004.

[15] E. N. Elnozahy and W. Zwaenepoel. Replicated distributed processes in manetho. In *In Proceedings of the Twenty Second Annual International Symposium on Fault-Tolerant Computing, FTCS-22*, pages 18–27, 1992.

[16] C. Engelmann and A. Geist. Super-scalable algorithms for computing on 100,000 processors. In *PROCEEDINGS OF ICCS*, pages 313–321. Springer, 2005.

[17] C. Engelmann, G. Vallee, T. Naughton, and S. Scott. Proactive fault tolerance using preemptive migration. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 252 –257, feb. 2009.

[18] T. Erlebach, H. Kellerer, and U. Pferschy. Approximating multiobjective knapsack problems. *Manage. Sci.*, 48:1603–1612, December 2002.

[19] G. E. Fagg and J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages

346–353, London, UK, 2000. Springer-Verlag.

[20] S. Fu and C. zhong Xu. Exploring event correlation for failure prediction in coalitions of clusters. In *in Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SCÕ07*, 2007.

[21] V. Getov, A. Hoisie, and H. J. Wasserman. Codesign for systems and applications: Charting the path to exascale computing. *Computer*, 44:19–21, Nov. 2011.

[22] R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. Panda, A. Lumsdaine, and J. Dongarra. Cifts: A coordinated infrastructure for fault-tolerant systems. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 237 –245, sept. 2009.

[23] W. Hoarau, P. Lemarinier, T. Herault, E. Rodriguez, S. Tixeuil, and F. Cappello. Fail-mpi: How fault-tolerant is fault-tolerant mpi? In *Cluster Computing, 2006 IEEE International Conference on*, pages 1 –10, sept. 2006.

[24] H.-Z. Huang, Z. Tian, and M. Zuo. Intelligent interactive multiobjective optimization of system reliability. In G. Levitin, editor, *Computational Intelligence in Reliability Engineering*, volume 39 of *Studies in Computational Intelligence*, pages 215–236. Springer Berlin / Heidelberg, 2007.

[25] K.-H. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, C-33(6):518 –528, june 1984.

[26] J. Hursey and R. Graham. Building a fault tolerant mpi application: A ring communication example. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1549 –1556, may 2011.

[27] J. Hursey, S. Hampton, P. Agarwal, and A. Lumsdaine. An adaptive checkpoint/restart library for large scale hpc applications. In *14th SIAM Conference on Parallel Processing and Scientific Computing*, 2010.

[28] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H. J. Epema. The grid workloads archive. *Future Gener. Comput. Syst.*, 24(7):672–686, July 2008.

[29] W. Jones, J. Daly, and N. DeBardeleben. Application resilience: Making progress in spite of failure. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, pages 789 –794, may 2008.

[30] R. Kumar and N. Banerjee. Analysis of a multiobjective evolutionary algorithm on the 0-1 knapsack problem. *Theor. Comput. Sci.*, 358:104–120, July 2006.

[31] J. Lee, S. Chapin, and S. Taylor. Reliable heterogeneous applications. *Reliability, IEEE Transactions on*, 52(3):330 – 339, sept. 2003.

[32] A. B. Nagarajan and F. Mueller. Proactive fault tolerance for hpc with xen virtualization. In *In Proceedings of the 21st Annual International Conference on Supercomputing (ICSÕ07*, pages 23–32. ACM Press, 2007.

[33] P. Nguyen, T. A. Simon, M. Halem, D. Chapman, and Q. Li. A hybrid scheduling algorithm for data intensive workloads in a map reduce environment. In *5th IEEE/ACM International Conference on Utility and Cloud Computing*, UCC 2012. ACM/IEEE, 2012.

[34] I. Raicu. *Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing*. VDM Verlag, Saarbrucken, Germany, 2009.

[35] I. Raicu, I. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1 –11, nov. 2008.

[36] T. A. Simon, P. Nguyen, and M. Halem. Multiple objective scheduling of hpc workloads through dynamic prioritization. In *High Performance Computing Symposium(HPCS'13),SpringSim 2013*. Society for Computer Simulation International/ACM, 2013.

[37] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[38] J. Stearley. Towards informatic analysis of syslogs. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 309–318, Washington, DC, USA, 2004. IEEE Computer Society.

[39] C. Valenzuela. A simple evolutionary algorithm for multi-objective optimization (seamo). In *Proceedings of the 2002 Congress on Evolutionary Computation. (CEC.)*, volume 1, pages 717 –722, may 2002.

[40] Y. Vorobeychik, J. R. Mayo, R. C. Armstrong, R. G. Minnich, and D. W. Rudish. Fault oblivious high performance computing with dynamic task replication and substitution. *Comput. Sci.*, 26(3-4):297–305, June 2011.

[41] L. Wolstenholme. *Reliability modelling: a statistical approach*. Chapman & Hall/CRC Press, 1999.