CAVS REPORT
MSU.CAVS.CMD.2009-R0008

Towards the robustness of high-performance execution of multiscale numerical simulation codes hosted by the Cyberinfrastructure of CAVS @ MSU

Project: MFERD/ICME/SRCLID
Project Leaders: Paul Wang, Mark Horstemeyer
Task 2: Cyberinfrastructure
Task 2 Leader: Tomasz Haupt
Task 2 Members: Ricolindo L. Carino, Anand Kalyanasndara, Florina M. Ciorba
External Collaborator: Ioana Banicescu

Prepared by **Florina M. Ciorba**
Center for Advanced Vehicular Systems
Mississippi State University
Mississippi State, MS 39762

Web site: http://www.cavs.msstate.edu

For public release.

TABLE OF CONTENTS

1. Background

The creativity of researchers is hampered by the intimidating complexity of computational systems (shared memory systems, distributed memory systems, single processor machines, multiple processors or multicores machines) and disciplines (physics, materials science, and solid mechanics), heterogeneity and multidisciplinary nature of the simulation codes, as well as the dispersion of codes, data, and information. Moreover, the rapid development of computer technology and commercial simulation tools has enabled researchers to solve more complex problems, as well as to obtain more details to enhance knowledge. The increased complexity has, however, increased significantly the computational cost, which in turn prevents scientists and engineers from combining the simulation tool with an optimization method, which is an iterative procedure that requires hundreds, or even thousands, of simulations to perform one design. In order to exploit the recent transformative research in materials science at the Center for Advanced Vehicular Systems at the Mississippi State University (involving multiscale physics-based predictive modeling, multiscale experiments and design), it is imperative to create a cyberinfrastructure that provides the necessary support in an effective and efficient manner.

In the framework of **Task 2** of the DOE SRCLID research project, the main objective of the cyberinfrastructure of CAVS @ MSU (Figure 2) is to develop a computational infrastructure supporting the multidisciplinary design optimizations. The cyberinfrastructure will result in the development of a ”community of practice” portal that allows the development and integration of multiscale physics-based materials models, multiscale experiments, and design for selected properties

and processes. The advent of large-scale computing systems organized into cyberinfrastructures enables the emergence of new classes of applications (e.g. large and very large scale simulations), which need to be handled carefully in order to harness efficiently the full capabilities of these resources. Moreover, the large-scale computing systems are unstable computing environments subject to multiple sources of uncertainties, have a high management complexity, and are comprised of computing resources of various types, quantity and availability. Therefore a successful cyberinfrastructure will need to manage effectively and efficiently the large-scale computing resources, provide mechanisms for efficient resource discovery (hardware, software and data), employ methods for flexible and reliable workflow allocation, and seamlessly integrate simulation programs (including multiphysics and multiscale simulations with uncertainty), CAD/CAE tools and various functional components, such as design of experiments, metamodeling, and design optimization.

The sequence of operations associated with using the material models, experiments and simulation programs via the cyberinfrastructure portal (CI-P) for the purpose of designing advanced materials with selected properties, can be modeled and represented by workflows. The CI-P is intended to be a multi-user portal, which will enable access of multiple users at a time. This will result in multiple workflows that will need to be handled simultaneously. Many software systems exist to support workflows in particular domains. Such systems manage tasks such as resource discovery, partially automated processing and integration between different functional software applications and hardware systems that contribute to the successful management of the underlying workflows. The cyberinfrastructure system will have to manage two types of workflows (Figure 1): service workflows (the sequences of operations performed via web services of the cyberinfrastruc-

ture portal) and scientific workflows (the sequences of steps performed for solving the scientific application of interest). Both types of workflows can be hierarchical - allowing complex tasks to be composed of simpler components. A *scientific workflow* is a combination of data and processes into a structured set of steps (sequential or repetitive), that implement semi-automated computational solutions of a scientific problem. Scientific workflows require support for large data flows, execution monitor and control (including workflow ad-hoc changes), execution in dynamic environments where resources are not know a priori and may need to adapt to unforeseen changes, parameterized and hierarchical execution of scientific applications tasks, with sub-workflows that can be created and destroyed as necessary, as well as support services for integration on large-scale computing systems (grid portals, web portals, autonomic computing systems). The scientific workflows have science domain specific requirements, such as materials science, physics, biology, etc.

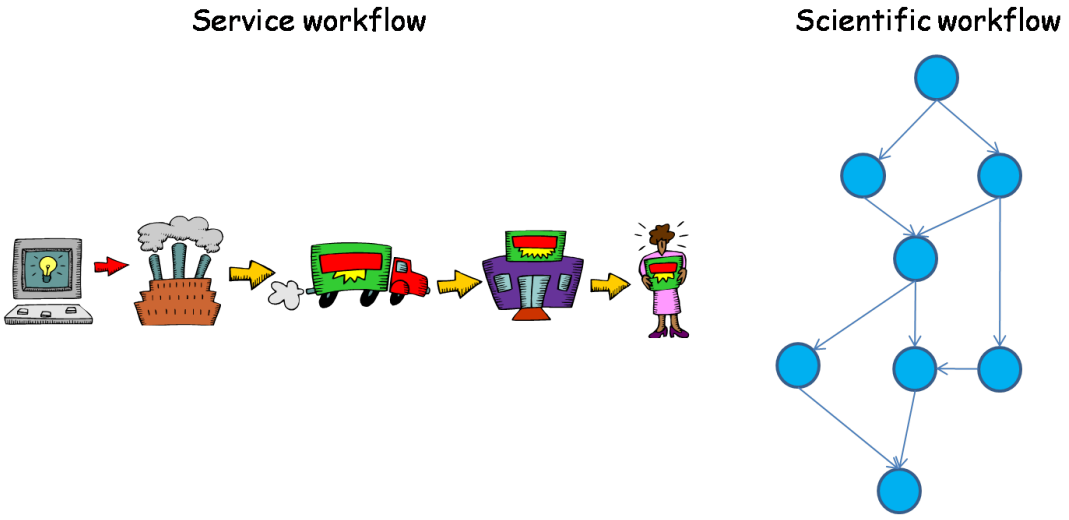

Figure 1. Workflows managed by the cyberinfrastructure

The cyberinfrastructure will enable the autonomous execution of multiple complex scientific

workflows on the high-performance computing resources it manages, while satisfying a set of desired goals (in the context of **Task 2.3** - Develop and deploy services for remote resource allocations and services for accessing remote file systems and databases). To ensure the autonomous execution of multiple complex computational workflows, advanced methods for scheduling, allocation and management are required. These methods must feature the following properties: (1) *self-optimization* (adaptive scheduling in high-performance heterogeneous distributed environments and adaptive resource management), (2) *self-healing* (robustness through monitoring of resource availability, fault detection and recovery, and conditional workflows), (3) *self-protection* (context-sensitive security), and (4) *self-configuration* (automatic detection of new hardware, software, and data components).

The design of the cyberinfrastructure will enable it to accommodate multiple users concurrently, each of which may desire to use a different range of services, and their individual goals may be competitive against the goals of the other users. The management of high-performance computing resources (onto which the services selected by the users – translated into jobs – are deployed for fast serial or parallel execution) must employ fairness such that the system-wide goals, as well as each individual user's goals, are satisfied. The system-wide goals can be complex, multidimensional and conflicting, and such goals are characteristic particularly to autonomic computing systems. The high-performance computing system managed by the cyberinfrastructure, in conjunction with the self-optimization, self-healing, self-protection and self-configuration properties, actually form an autonomic high-performance computing system, or autonomic computing system, for short, as illustrated in Figure 3. Due to these properties, autonomic computing systems (ACS) become

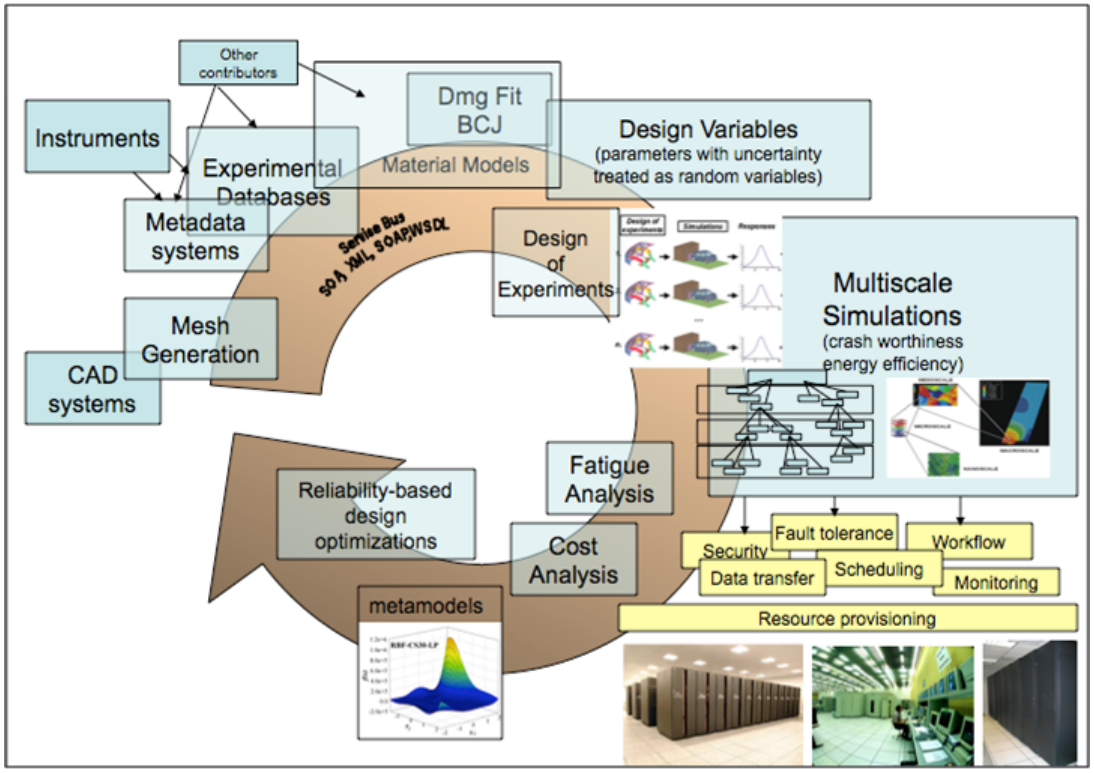

Figure 2. The vision - the cyberinfrastructure of CAVS @ MSU will integrate distributed databases of engineering knowledge and computational workflows for multi-objective, multiscale design optimizations under uncertainty.

capable of dealing with uncertain and shifting operating conditions that can be known only at runtime.

The research work in this report addresses the 'self-optimization' (*flexible* workflow scheduling methods capable of *adapting* to changes in the resources' availability and capability) and 'self-healing' (*reliable* workflow scheduling methods capable of recovering from failures) properties of the cyberinfrastructure autonomic computing system (cyber-ACS). The goal of this research work is to employ state-of-the-art workflow scheduling methods, to enable the autonomous execu-

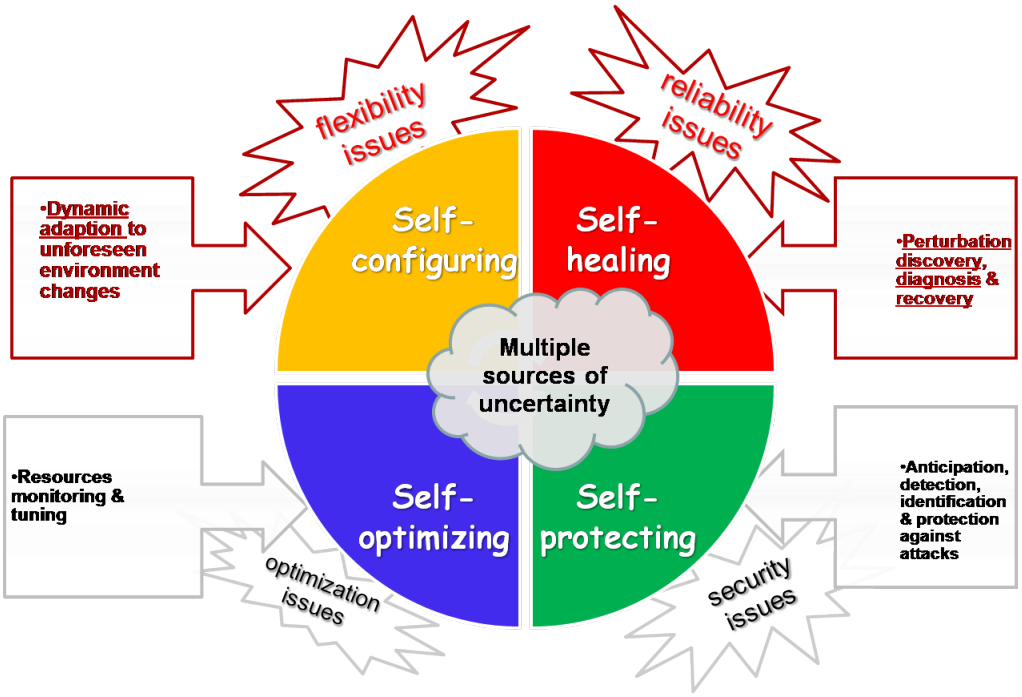

Figure 3. Cyber-ACS: the cyberinfrastructure autonomic computing systems – characteristics and issues

tion of multiple complex computational workflows, while meeting the system-wide set of desired goals. Towards achieving this research goal, a theory of robustness for such methods is needed, including definitions and analyses of robustness, diversity, redundancy, and optimality and their relationship to one another [11]. **The research work in this report is the first step towards a robustness theory for autonomic complex workflow scheduling methods for high-performance multiscale numerical methods.**

In order to formulate a robustness theory for workflow scheduling methods on the cyber-ACS, we begin defining the possible causes (and their source) responsible for the computational per-

formance degradation of executing such complex scientific workflows. In this work we focus on enabling the flexible and reliable execution of scientific workflows, and the fishbone diagram illustrated in Figure 4 identifies multiple causes impacting the flexibility and reliability of their execution, having the ultimately the same effect - degradation in the computational performance.

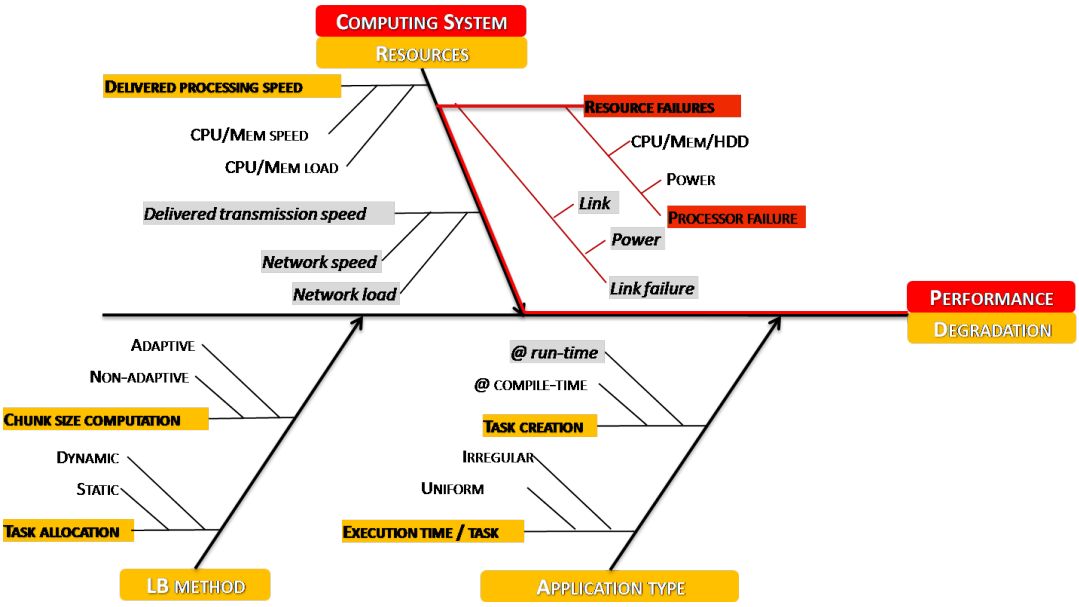

Figure 4. The relationships between the multiple causes and their effect on the computational performance of the execution of scientific workflows

With regards to the scientific workflow scheduling methods, in this work we focus on dynamic loop scheduling (DLS) methods, which have been demonstrated to be a powerful approach towards improving the performance of scientific and engineering applications, expressed as complex workflows, via load balancing. DLS methods provide application-level load balancing of repetitive computations, expressed as loop iterates, with the goal of maximizing application's performance

on the underlying system. These methods use performance information about the application execution (for which irregularities change over time) at run-time. In addition, many DLS methods are based on probabilistic analyses, and therefore account for unpredictable variations of application- and system-related parameters. Scheduling scientific and engineering applications in large-scale distributed computing systems (possibly shared with other users, as in the case of the cyberinfrastructure) makes the problem of scheduling complex workflows, with the goal of achieving the best possible performance, all the more challenging. Therefore, a hierarchical approach is needed for obtaining satisfactory performance using DLS.

The *distributed*, *heterogeneous* and *shared* nature of the computational resources managed by the cyberinfrastructure, greatly increases the chances of failures (such as processor or link failure) occurring during the execution. In this report, we present the theoretical results of investigating the robustness of three hierarchical DLS methods (FAC, WF and AWF), for scheduling applications expressed as irregular computational workflows in large-scale heterogeneous distributed systems, with respect to variations of two system parameters: *load* and *resource failures*. In this work, our focus is to devise robustness metrics for a class of hierarchical task scheduling algorithms, for a class of applications as well. Using a general methodology proposed by other researchers for devising robustness metrics, we propose two robustness metrics for three hierarchical DLS algorithms. These metrics will be used in conjunction with the hierarchical DLS methods, to provide quantitative and qualitative information such as: performance level, correctness and quality of execution, of scientific applications expressed as complex workflows, in large-scale heterogeneous systems, such as the cyber-ACS.

2. Introduction

The advent of large-scale computing systems enabled the emergence of new classes of applications, in order to harness the great capabilities of these resources. Researchers and scientists from various fields are interested in the accurate modeling and simulation of various complex phenomena. These phenomena are of great importance for the validation and calibration of new theoretical simulation models at different length scales, in areas such as mechanics, materials, physics, chemistry, biology, applied mathematics, and others. These simulations are often routines that perform repetitive computations (in the form of DO/FOR loops) over very large data sets, which if not properly implemented, may become a bottleneck for advancing the research in deriving new state-of-the-art models. The number of repetitive computations (iterations) in these codes is not always constant. Moreover, their nature (or computational requirements) may be irregular, in the sense that one computation may take more time than others, depending on the simulation. The resources in a large-scale system are widely distributed and highly heterogeneous. Consequently, they are usually shared among multiple users, and their availability cannot always be guaranteed or predicted. Hence, the *quality* and *quantity* of resources available to a single user changes continuously.

In this work, dynamic loop scheduling (DLS) techniques are considered to be the key solution for achieving and preserving the best performance of these applications in such environments. Herein, it is considered that a '*loop iteration*' (or a chunk of loop iterations) with variable execution time refers to a '*task*' (or a chunk of tasks, among many others within a loop of tasks) with variable execution time. A considerable number of static and dynamic loop scheduling algorithms has

been proposed for parallel loops, for loops with dependencies, as well as for loops with regular or irregular execution times per iteration, respectively. The use of static scheduling algorithms has been limited, due to the fact that nowadays computing systems can be variably loaded, rendering the initial assumptions of a static schedule invalid. A large number of DLS techniques have been developed over time. A comprehensive description has been given earlier in a survey by Hurson et al. in [6] and in the relevant literature after that.

Dynamic loop scheduling methods provide two alternative approaches, *non-adaptive* and *adaptive*, for achieving a good load balancing on variably loaded resources, as well as for executing tasks with varying execution times. Most of the techniques described in [6] are based on probabilistic analyses and are non-adaptive. Other non-adaptive techniques that were not mentioned in the survey above include fractiling [4] and weighted factoring [5]. Subsequent efforts gave birth to more elaborate techniques, called adaptive, and a few examples are given in [9][12][17]. Most of the above adaptive methods are based on probabilistic analyses, and use a combination of run-time information about the application and the system, in order to predict the system capabilities for the next computational assignments, or to estimate the time future tasks will require to finish execution, in order to achieve the best allocation possible for optimizing application performance via load balancing. In this paper, we concentrate on two non-adaptive techniques, factoring [3] (FAC) and weighted factoring [5] (WF), and one adaptive technique, adaptive weighted factoring [10] (AWF). The techniques considered herein use probabilistic analyses to dynamically compute the size of chunks at run-time, such that they are executed before their optimal time with high probability.

Motivation Scheduling applications on large-scale shared environments makes the task of devising appropriate DLS methods quite challenging. The challenge is even higher on large-scale *heterogeneous* systems, where chances of faults (for example processor/link failure) are high. Therefore, a viable DLS-based solution should employ an approach based on hierarchical management and mechanisms to ensure their robustness. The choice of considering the three FAC, WF and AWF DLS methods is due to the fact that they are *inherently robust*, because they are designed in such a way to address unpredictabilities in the application and the system. Until recently, the robustness of resource allocations or task scheduling algorithms was addressed individually for a single resource allocation algorithm, or a single task scheduling method, or even a single application. Ali et al. [18], proposed a general methodology for devising robustness metrics for resource allocations. The goal of this work is to use this general methodology for devising robustness metrics for a class of task scheduling algorithms, and possibly for a class of applications as well.

Contribution In order to extend the *scalability* of the three DLS methods (FAC, WF and AWF) to large-scale heterogeneous platforms, we propose using processor groups as the hierarchical management approach. Hierarchical DLS methods based on processor groups have been shown to achieve better performance than centralized DLS methods [13][14][16]. Furthermore, in the hierarchical approach, the groups of processors are dynamically selected such that the physical structure of the underlying execution environment is well and easily captured. This leverages the best possible performance of the application on the large-scale platform. As a first step towards devising mechanisms for robust DLS, we propose two *robustness metrics* to quantify the robustness of the FAC, WF and AWF algorithms against variations of two system related parameters:

load and *resource failures*. The main contribution of our work is using the methodology proposed by Ali et al. [18] to propose these *two* robustness metrics for *three* hierarchical DLS algorithms, which can be used to schedule a very important and large *class* of applications: those that have *irregular* tasks (expressed as irregular workflows). These metrics are used in conjunction with the hierarchical DLS methods to provide quantitative and qualitative information regarding the level of performance, and the quality of execution for executing such applications in uncertain large-scale heterogeneous systems. The standalone metrics are no better than any other performance measurement metrics (e.g. makespan, communication cost, resource utilization). However, their use is mandatory towards achieving robustness for the hierarchical DLS algorithms running on such uncertain systems.

3. Overview of DLS methods

FAC, WF and AWF are intended for serial applications that contain straightforward parallel loops (1-d loops) or nested parallel loops (2-d loops), as illustrated in Figure 5, using Fortran 90 notation. Each loop iterate (iteration) is considered to be a task. Parallel loops have no depen-

```
a) 1D-loop          b) 2D-loop
...                 ...
DO I=1,N            DO J=1,M ! J-LOOP
... I-ITERATE       ... PART A OF J-ITERATE
END DO              DO I=1,N(J) ! I-LOOP
...                 ... I-ITERATE of J-ITERATE
                    END DO ! I
                    ... PART A OF J-ITERATE
                    END DO ! J
```

Figure 5. Target applications model

dencies among their iterates, which can be executed in any order or even simultaneously without affecting the correctness of the computations.

Assuming a total of N tasks and P processors, let T_j be the execution time of task a_i on processor m_j. We assume that T_j ($1 \leq i \leq N$ and $1 \leq j \leq P$) are *independent random variables* ($i.r.v.$), and we further assume that T_j are *identically distributed* ($i.d.$) for all tasks executed on processor m_j. Denote the *mean* and the *variance* of T_j by μ_j and σ_j^2, respectively. Then, the average execution time $ET_j(k)$ of a chunk size of k tasks by processor m_j is $k\mu_j$, with standard deviation $\sqrt{k}\sigma_j$. When the chunk size k is large, by the Central Limit Theorem, we may treat the execution time as a normal random variable. The execution time of a batch of P chunks of size k_j ($1 \leq j \leq P$) tasks by processors $m_1, \ldots, m_P$ is the largest order statistic $\max\{T_j(k_j), 1 \leq j \leq P\}$. The optimal execution time for the N tasks is: $T_{opt} = k_1\mu_1 = \ldots = k_P\mu_P$, with $k_1 + \ldots + k_P = N$. The optimal time can be achieved *only* if the task execution times are nonrandom.

Based on the processor speeds and knowledge of the mean and variance of task execution times, the DLS methods considered herein can be distinguished as follows.

3.1. Equal speed processors with known μ_j and σ_j^2

The tasks' execution times are *independent identically distributed random variables* ($i.i.d.r.v.$) with known μ_j and σ_j^2, regardless of which processor executes them. FAC [3] was devised for this situation. In FAC, the tasks are scheduled in batches, where the size of a batch is a fixed ratio of the unscheduled tasks, and the batch is divided into P equal size chunks. The ratio is determined from a *probabilistic analysis* such that the resulting chunks have a high probability of finishing *before*

the optimal time.

3.2. Weighted processors speeds with known μ_j and σ_j^2

In most situations, it is unrealistic to assume that all processors have equal speed. Therefore, the execution time of a task for different processors will have different μ_j and σ_j^2. The execution times of all tasks are *independent random variables* ($i.r.v.$), but not identically distributed *unless* they are executed by the same processor. WF [5] was devised for this situation, and incorporates information about the relative processor speeds in computing chunk sizes, where these speeds are assumed to be fixed throughout the execution of the loop.

3.3. Weighted & adapted processors speeds with known μ_j and σ_j^2

A number of methods that generate *adaptive* size chunks have evolved from FAC and WF. The requirement for the processor speeds in WF is relaxed in AWF, a method developed to be utilized in time stepping applications [10] (and references therein). In AWF, the processor weights are initially set to unity for the first time step. The execution times of chunks during a time step are recorded, and the data is used to *adapt* the processor weights at the end of each time step. The AWF, however, does not adapt to any load imbalance that occurs *during* the current step.

Generally, the goal of all DLS methods is minimizing the total execution time, T_{PAR}. The total execution time given by a DLS that uses dynamic load balancing is expressed as the finishing time of the processor executing the *last* task. Mathematically, assuming $\mathcal{ET}_j$ to be the finishing time of all tasks a_i executed by processor m_j, the total parallel execution time can be written as:

$$T_{PAR} = \max(\mathcal{ET}_j, 1 \leq j \leq P) \quad (1)$$

T_{PAR} is minimized when the coefficient of variation ($c.o.v.$) of all processor finishing times is very low. The $c.o.v.$ is defined as: $c.o.v. = \frac{\sigma_{\mathcal{ET}_j}}{\overline{E}_j}$ with $\overline{E}_j$ the mean of these times and $\sigma_{\mathcal{ET}_j}$ their variance. A $c.o.v.$ ratio close to 0 indicates that all processors have similar finishing times, suggesting a good load balancing, whereas a ratio closer to 100% indicates a high variability of the finishing times, suggesting an unbalance in the computational loads that could stem from the non-uniformity of task execution times, variation of processors load, and/or other forms of system interference during the parallel execution.

4. Hierarchical management system model

Most loop scheduling methods are developed assuming a central ready work queue of tasks (central management approach), where idle processors obtain chunks of tasks to execute. This approach is ideally suited for shared memory architectures. On message-passing distributed systems, however, these methods are implemented using a *foreman-worker* strategy. The scheduling decisions are centralized in the foreman, but since there is no centralized memory, the work queue of tasks may be replicated on all processors, or partitioned among the processors. In either case, accessing the foreman may become a bottleneck when a large number of workers attempt to simultaneously communicate with it. A well-known disadvantage of centralized approach, especially for message-passing systems, is its limited practical scalability as the number of processors increases.

To address this bottleneck, a useful modification of the centralized strategy, especially for parallel applications with a large amount of communication requirements, is using a partition of the

processors into *disjoint processor groups*, which concurrently execute independent parts of the problem while interacting if, and when, necessary. This two-level (hierarchical) programming strategy is achieved by utilizing *multiple-foremen* (see Figure 2).

The performance of DLS methods based on the hierarchical management approach has been shown to be better than that of the centralized management approach [13][14][16]. In the multiple-foremen strategy, the processors of a group are *dynamically* selected such that the physical structure of the underlying platform is well and easily captured. For instance, disjoint groups of processors may correspond to disjoint clusters (or grid sites), and their size are determined by the clusters' (or grid sites') sizes. This dynamic group formation assists the DLS methods to leverage the best possible application performance on the large-scale platform.

In the hierarchical management approach, the set of N processors are initially divided into *processor groups*, the group sizes being chosen to match the *degree of concurrency* exhibited by the target application, as well as the *structure* of the underlying platform. A large application may require hundreds of processor groups, and a processor group may contain hundreds of processors. The processors may join and leave the groups voluntarily, or because failures (processor/link) have occurred. Each group executes the foreman-worker strategy on a specified number of application tasks (see Figure 6). All groups execute the multiple-foremen strategy in order to execute all tasks of the target application (see Figure 2). Each foreman processor maintains for its group, the ratio of the cost of remaining tasks to the number of worker processors, and periodically reports the ratio to a manager processor (designated at group formation). The frequency of reporting these ratios to the manager depends on the DLS method used. Upon detecting large differences in the

reported ratios, the manager initiates the transfer of workers *between* groups to balance the ratios (see Figure 2 - right), while remaining truthful to the underlying architecture. In the hierarchical management setting, DLS is accomplished via load balancing at each of the two levels: foreman-worker and multiple-foremen. A more elaborated description of the interactions and coordination of processors using the hierarchical management setting is given in [13].

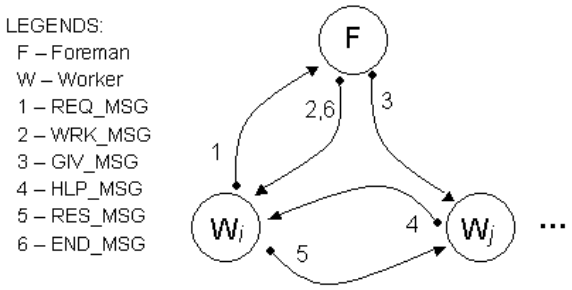

Figure 6. Centralized management system

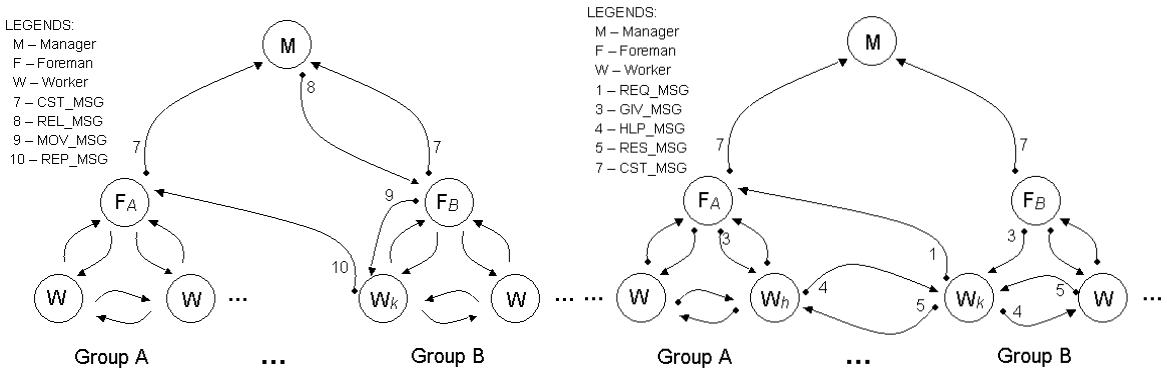

Figure 7. Hierarchical management system

5. Robustness – definition

Robustness is an emergent multifaceted phenomenon in advanced computing systems. Depending on the context, the term '*robustness*' can refer to software robustness (e.g. operating system, API), system robustness (e.g. embedded systems), scheduling robustness (e.g. resource allocation), algorithm robustness (e.g. computational algorithms, scheduling algorithms) etc.

The multitude of aspects of robustness, yielded many definitions. It is very difficult (if even possible) to include all these aspects in one universally valid definition. Nevertheless, for resource allocations Ali et al. gave the following definition in [18]: "*A resource allocation is defined to be robust with respect to specific system performance features against perturbations (uncertainties) in specified system parameters if degradation in these features is constrained when limited perturbations occur.*" This is the closest to application scheduling definition of robustness we could find.

In the last decade, a trend has emerged towards the design of robust methods for resource allocation, application scheduling, etc. However, another similar trend, that appeared approximately at the same time, concerns the design of methods that address the issues of fault-tolerance and resilience [20][21] (and references therein). Even though the robustness of various resource allocations and scheduling algorithms has been the focus of many researchers in the past years, it is still considered to be an emergent property leaving room for much research in the years to come. This is also supported by the multitude of factors that reflect or impend the robustness of such methods.

We are interested in finding the answer to the following question "What is the difference be-

tween robustness and fault-tolerance?" There is no clear distinction between the two research trends. Following a careful overview of the existing literature in both trends: robustness and fault-tolerance/resilience in the context of resource allocation and scheduling algorithms, as well as other studies on the robustness of complex [1] or immune [19] systems, we reached the conclusion that the concepts of '*fault-tolerance*' and '*resilience*' are encompassed by the more generic concept of '*robustness*'. The relationship between the two trends, as given by Gribble in [1]: "*complex systems* (in our case advanced scheduling algorithms) *must expect failures and plan for them accordingly*." comes to support our conclusion above. In addition, the statement from [18]: "... *the flexibility of a resource allocation ... is described as the quality of the resource allocation that can allow it to be changed easily into another allocation of comparable performance when system failures occur.*" is yet another illustration of the connection between robustness and fault-tolerance, particularly for resource allocations. Hence, we can safely infer that the flexibility of a scheduling algorithm reflects its robustness against unpredictable failures in the system.

There are several generic guidelines towards the design of robust methods, regardless of the parameters whose perturbations (uncertainties) have to be asserted. Although some of these guidelines are not specifically intended for resource allocations or scheduling algorithms, they are still applicable in these contexts. One such illustrative guideline given in [1], suggests to avoid designing systems that attempt to gain robustness only by means of predictable information, as they will be prone to fragility.

Designing specifically for robustness, to its full extent, is however not yet possible. Designing a scheduling algorithm with the goal of achieving robustness gives no guarantee that the algorithm is

more robust than an algorithm designed without that goal. Robustness benchmarks (or metrics) can, therefore, be useful to measure *how* a scheduling method reacts to possible erroneous inputs or environmental factors. Accordingly, robustness metrics should employ mechanisms for *detecting* and *identifying* such erroneous parameters. Ultimately, such metrics can be very useful and efficacious for estimating the robustness of the method of interest against certain perturbation parameters. In this work we focus on designing robustness metrics that *model* and *estimate* the robustness of DLS algorithms on large-scale realistic platforms against various perturbation parameters.

6. Designing robustness metrics using FePIA

This section describes a general methodology for designing robustness metrics for scheduling algorithms. The target systems are large-scale and consist of distributed and heterogeneous processing resources. We use the FePIA (features-perturbation-impact-analysis) procedure [18] (and references therein) for deriving metrics that model and estimate the robustness of three DLS algorithms against various perturbation parameters. The FePIA procedure consists of four general steps:

Step 1. *Identify the performance features.* Describe quantitatively the requirements that make the algorithm robust: determine the performance features (e.g. makespan, average execution time per processor or per task) that should be limited in variation to ensure that the robustness requirement is met.

Step 2. *Identify the perturbation parameters.* Determine all uncertainty parameters (application, system or environment related) whose values may impact the performance features selected in Step

1.

Step 3. *Identify and clarify the impact of perturbation parameters* (in Step 2) *on performance features* (in Step 1).

Step 4. *Identify the analysis to determine the robustness.* Determine the smallest collective variation in the values of perturbation parameters identified in Step 2, which will cause any of the performance features identified in Step 1 to violate their corresponding acceptable variation, i.e., determine the robustness radius for each type of impact determined in Step 3. This gives the collective *degree of robustness* (as the minimum of all robustness radii) of the DLS algorithm against the parameters selected in Step 2.

6.1. Towards robust DLS algorithms

We outline here the proposed approach for DLS-based task scheduling on large-scale platforms using hierarchical management. The N tasks are assumed to be independent. The goal is scheduling these tasks onto the set of P processors of the large-scale heterogeneous distributed system, while minimizing the total parallel execution time (or *makespan*) T_{PAR}. A minimum T_{PAR} is achieved via dynamic load balancing, using processor speeds (which in the case of AWF are periodically adapted) and hierarchical management. Each processor in a group executes a set of tasks (called chunk) at a time. Each task is executed in a non-preemptive fashion, i.e., no other tasks of higher priority will suspend it. The same holds for the execution of a chunk, or for all chunks during a single time-step. Table 1 summarizes the notations we used in the following sections.

We define the performance features of interest for the three DLS algorithms: *the processors*

N	total number of tasks
N^{resch}	# of tasks that need to be *rescheduled*
a_i	i-th task, $1 \leq i \leq N$
P	total number of processors
m_j	j-th processor, $1 \leq j \leq P$
T_j	*execution* time of task a_i on m_j
T_{ij}^{W2F}	*communication* time between m_j and its foreman for executing a_i
T_{ij}^{W2W}	*communication* time between m_j and any other workers for executing a_i (processor regrouping)
$\mathcal{ET}_j$	*finishing* time of all tasks computed by m_j
T_{PAR}	total parallel execution time for the N tasks
$\lambda = [\lambda_1 \dots \lambda_P]^T$	vector of processors *load* (= system load)
$\mathbf{F} = [\mathbf{f}_1 \dots \mathbf{F}_P]^T$	resources status vector (active/*failed*)
$\Phi = \{\phi_1, \dots\}$	set of *performance* features
$\Pi = \{\pi_1, \dots\}$	set of *perturbation* parameters
τ_1, τ_2, τ_3	tolerance factors for performance features
$r_{DLS}(,)$	robustness radius
$\rho_{DLS}(,)$	robustness metric

Table 1. Notation

finishing time, $\mathcal{ET}_j$, the total parallel time, T_{PAR}, and the number of tasks that need to be rescheduled, N_{resch}. These performance features should be limited in variation under certain application, system or environment related parameters perturbations. In the context of DLS, the perturbation parameters include variations of the following: irregularities of application computational requirements, *system availability due to unforeseen loads* (processors' delivered computational speed when shared among multiple users), network latency (delays in the communication speed due to network congestion), and *resource reliability* (caused by processor or network failures). Commonly, *all* these perturbation parameters vary over time and cannot be accurately predicted before the parallel execution. A robust DLS algorithm must adapt to any variations of these perturbation parameters, and yield performance parameters that vary in a constrained manner. Designing robustness metrics that incorporate *all* these parameters is very challenging [18].

In this work, we devise metrics for measuring the robustness of three DLS algorithms (FAC, WF and AWF) against perturbations in two system related parameters: *system load* and *resource failures*.

6.2. Flexibility against perturbations in system load

In order to measure the robustness of FAC, WF and AWF against *system load* perturbations, we use the FePIA procedure to devise the appropriate metric. We make the following assumptions with respect to perturbations in the load of the large-scale system during run-time:

A.1 variations of individual worker loads are mutually independent

A.2 individual worker loads may or may not occur simultaneously

A.3 DLS has load variation detection and monitoring mechanisms

In heterogeneous large-scale systems, *unforeseen system load variations* are to be expected, and therefore we require that the individual finishing time $\mathcal{ET}_j$ of processor m_j be robust against them. According to eq. (1), this requirement translates into the requirement that the total parallel execution time T_{PAR} be robust against such variations, as well. We do not assign different "weights" to the impact of one processor's finishing time or another's, due to the fact that each processor is equally likely to impact T_{PAR}. Assuming system load variations, the *actual* finishing time $\mathcal{ET}_j$ of processor m_j must be calculated considering the effects of errors in the estimation of the processor's load variation, and *must not* exceed $\tau_1 (> 1)$ times its estimated value $\mathcal{ET}_j^{orig}$, where τ_1 is a tolerance factor reflecting the robustness. The FePIA procedure for this analysis is outlined below.

Step 1. Let $\Phi = \{\phi_1\}$, where $\phi_1 = \mathcal{ET}_j$ and $1 \leq j \leq P$ be the performance features set. The individual finishing time, $\mathcal{ET}_j$, of processor m_j is the sum of *computation* times T_j, of all tasks a_i executed by m_j, and the sum of *communication* times T_j^{W2F}, between processor m_j and its corresponding foreman, and the sum of *communication* times T_j^{W2W}, between m_j and any other worker processor. Mathematically, for all $\{\text{tasks } i | a_i \text{ executed on } m_j\}$, this is written as:

$$\mathcal{ET}_j = \sum_{i,j}^{N,P} \left(T_j + T_j^{W2F} + T_j^{W2W} \right) \quad (2)$$

Step 2. Let the perturbation parameters set be $\Pi = \{\pi_1\}$, where $\pi_1 = \lambda_j$ and $1 \leq j \leq P$. For our analysis, we consider the perturbation parameter to be λ_j, the *individual* load of processor m_j. Vector λ contains the load values of all processors in the target system. DLS *initially assumes* that

the system has λ^{orig} load. The value of λ^{orig} can be usually determined by executing the first batch of chunks, as determined by the original factoring rules and their subsequent evolution. The initial load of m_j is λ_j^{orig}, found at the j-th position in the λ^{orig} vector.

Step 3. As explained in the previous subsection, in order to determine the impact of λ_j over $\mathcal{ET}_j$, we analyze individually, for all processors, their finishing time given their *own* load. Each actual finishing time is expected to vary with according to λ_j. This is denoted as $\mathcal{ET}_j(\lambda_j)$ – the actual execution (computation and any communication associated with it) time of all tasks a_i assigned to m_j, in the presence of load variation on m_j, as indicated by λ_j. Mathematically, for all $\{$tasks $i|a_i$ executed on m_j under varying load $\lambda_j\}$, this is written as:

$$\mathcal{ET}_j(\lambda_j) = \sum_{i,j}^{N,P} \left(T_j(\lambda_j) + T_j^{W2F}(\lambda_j) + T_j^{W2W}(\lambda_j) \right) \quad (3)$$

Step 4. At this step, for every parameter in Φ, we need to define the boundary values of the $\pi \in \Pi$ under consideration. A key role for deriving appropriate boundary relationships, is played by the possibility that the perturbation parameter is a continuous or a discrete variable. For our analysis, Π has only one parameter: $\pi_1 = \lambda_j$. It is really a matter of taste to consider λ_j a discrete or a continuous variable. Traditionally, the load of a processor for task scheduling in heterogeneous systems is measured either as the number of processes in the processor run-queue [2], or as the processor delivered speed [21]. The first measure renders λ_j a discrete parameter, taking integer values larger than or equal to 1. The second measure renders λ_j a continuous parameter, measured as availability percentage of the particular processor (in our case m_j) for computing our tasks, usually taking values of 60%, 80% or 95%. For one of their example systems considered in [18] for resource allocation, Ali et al. assumed that the system load λ is a discrete variable (measured

in units of objects per data set), and for simplicity reasons treated it as a continuous variable.

In this work, we consider λ_j a *continuous* variable that measures the availability of processor m_j in %. The percentual availability of a procesor expresses its delivered computational speed, which encompasses all three effects of applications' requirements, hardware capabilities and network speed in one. The boundary values of λ_j must satisfy the following boundary relationships:

$$\left\{\lambda_j \in \left\langle \lambda_j', \lambda_j'' \right\rangle \mid \left(f_1(\lambda_j') = \beta_1^{\min}\right) \wedge \left(f_1(\lambda_j'') = \beta_1^{\max}\right)\right\} \quad (4)$$

The tolerable variation interval for the performance feature of interest, i.e., $\mathcal{ET}_j$, is given by $\langle \beta_1^{\min}, \beta_1^{\max} \rangle$. Even though it is assumed the processor load will be λ_j^{orig}, this value might differ in practice, due to inaccuracies in load estimations or unforeseeable changes in the environment. The tolerable increase in the *actual* finishing time, $\mathcal{ET}_j$, of processor m_j, considering the effects of errors in the estimation of variations of λ_j, cannot exceed $\tau_1 (> 1)$ times its estimated value $\mathcal{ET}_j^{orig}$. Then boundary relationships for this analysis are:

$$\left\{\lambda_j \in \left\langle \lambda_j', \lambda_j'' \right\rangle \mid \left(\mathcal{ET}_j(\lambda_j) = \tau_1 \mathcal{ET}_j^{orig}\right) \wedge (1 \leq j \leq P)\right\} \quad (5)$$

Next, we need to define a robustness radius, which is the *largest* increase in processor load in any direction (for any combination of processor load values) from the assumed value, that does *not* cause any tolerance interval violation for the execution time of all tasks a_i assigned to m_j. To define the robustness radius we need to choose which norm will give us the smallest variation in the system (and ultimately processor) load. The choice of a particular norm depends on the DLS algorithm (and target environment), for which we desire to measure the robustness. Another aspect to be considered when choosing the norm is the actual nature of the selected perturbation parameters.

According to the nature of λ_j, we believe that a more intuitive norm to use for determining the robustness radius would be the ℓ_1-norm. We can write the robustness radius using the ℓ_1-norm as follows:

$$r_{DLS}(\mathcal{ET}_j, \lambda_j) = \max \ \|\lambda_j - \lambda_j^{orig}\|_1 \ s.t. \ \mathcal{ET}_j(\lambda_j) = \tau_1 \mathcal{ET}_j^{orig} (6)$$

Finally, the robustness metric is the minimum of all robustness radii:

$$\rho_{DLS}(\Phi, \lambda_j) = \min\left(r_{DLS}(\phi_1, \lambda_j)\right) \ \forall \ \phi_i \in \Phi \ (7)$$

The *robustness metric* indicates that if the difference between the value of the actual system load and the value of the estimated system load is no larger than $\rho_{DLS}(\mathcal{ET}_j, \lambda_j)$, then the actual finishing time for m_j will be at most τ_1 times the estimated finishing time $\mathcal{ET}_j^{orig}$ value. An acceptable value for τ_1 was proposed in [18] to be 1,2. Depending on the nature of the application to be scheduled, one can select a larger value (for instance if the quality of the result of the computation is more important than the speed of getting it) or a smaller value (as in the case of real-time applications). For this analysis, $\rho_{DLS}(\mathcal{ET}_j, \lambda_j)$ is the robustness metric of the "DLS" algorithm in general with respect to the each processor's individual finishing execution time against perturbations in the processor load. And therefore, the robustness metric of the total parallel time T_{PAR} against variations in the total system load would be

$$\rho_{DLS}(T_{PAR}, \lambda) = \min(\rho_{DLS}(\mathcal{ET}_j, \lambda_j)), 1 \leq j \leq P \ \ (8)$$

This robustness metric can be used to determine the impact of system load variations on the performance of the three DLS, and differentiate them according to their flexibility against variations in the system load. Figure 8 illustrates two possible scenarios. The first scenario, describes the situation when the three DLS methods perform similarly in terms of performance, with the difference

being in the variation in system load that each of them is able to capture. Based on this scenario 1 and Figure 8, one should choose the DLS method that has the lowest impact on DLS performance and can handle the largest variation of Λ, which in this case is AWF. The second scenario, describes the situation when for the same captured variation of system load, the three DLS methods perform differently in terms of computational performance. For this scenario, one should choose the DLS method that has the lowest impact on DLS performance, regardless of the variation interval of Λ, which is AWF again.

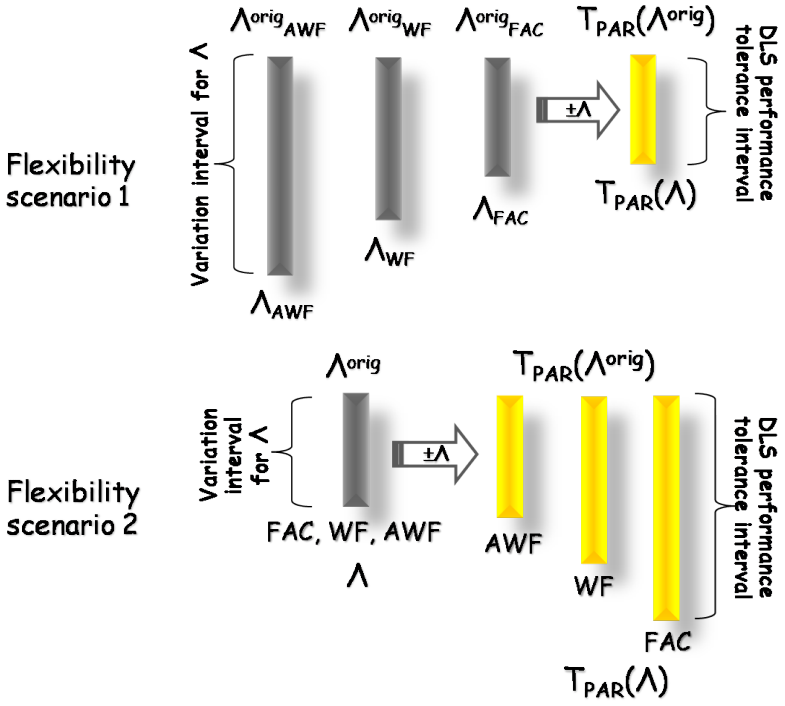

Figure 8. Possible scenarios to determine the flexibility of FAC, WF and AWF

6.3. Reliability of DLS against resource failures

Motivated by Gribble's suggestion [1] that the design (of scheduling algorithms) should plan for failures, we propose a metric to measure the robustness of DLS algorithms against *resource failures*. As for the previous metric, we again use the FePIA procedure.

Assuming that system is un-safe (i.e., failures are to be expected) we require that both, the *number of tasks to be rescheduled*, N^{resch}, and the *total parallel time*, T_{PAR}, given by the DLS algorithm, be robust against that. In the event of resource failures, N^{resch} *must not* exceed $\tau_2\%$ of the total number of tasks N, and T_{PAR} *must not* exceed $\tau_3(> 1)$ times it's estimated value T_{PAR}^{orig} (computed under the assumption that the system is completely safe).

In the presence of failures, the DLS algorithm must be able to reschedule the tasks that were assigned to the failed processors, as well as reschedule other tasks if necessary (for instance, to preserve load balancing on the remaining processors). To reduce the complexity of the analysis, we make the following simplifying assumptions:

A.4 only resources (e.g. network link, processor) associated with worker processors fail

A.5 resource failures occur simultaneously

A.6 resource failures are mutually independent

A.7 resource failures are permanent and

A.8 the DLS algorithm has fault-discovery and fault-recovery mechanisms

These assumptions can be relaxed in order to deliver more optimal and general robustness metrics. However, in order to ensure the accuracy of a *new* robustness metric, one should start with simple realistic assumptions. It is also well known that designing fault-tolerant scheduling algorithms to meet strict performance criteria is not at all trivial. As such, in the last decade it became an important research area in its own right, which is currently very active. Since there is ongoing work towards designing fault-tolerant DLS algorithms based on hierarchical management, we intend to integrate them with the robustness metrics proposed in this work. The FePIA procedure for this analysis is elaborated as follows.

Step 1. In contrast to the first analysis, the set of performance features has two elements in this case: $\Phi = \{\phi_1, \phi_2\}$, where $\phi_1 = N^{resch}$ and $\phi_2 = T_{PAR}$.

Step 2. In order to identify which processors fail, we use the approach proposed by Ali et al. in [18]. Thus, let $\mathbf{F} = [\mathbf{f}_1 \mathbf{f}_2 \dots \mathbf{f}_P]^T$ be the vector containing the live status of all resources, defined as:

$$\mathbf{f}_j = \begin{cases} 1 & if \text{ processor/link } \mathrm{m}_j \ \ failed \\ 0 & \text{otherwise} \end{cases} \qquad 1 \leq j \leq P \qquad (9)$$

The original value of $\mathbf{F}$ is expressed by $\mathbf{F}^{orig} = [0\ 0 \dots 0]^T$, indicating that initially all resources are alive. The perturbation parameters set in this case is $\Pi = \{\pi_1\}$, where $\pi_1 = \mathbf{F}$.

Step 3. In order to determine the impact of Π over Φ, we need to determine separately each of the two relationships:

$$\phi_1 = f_{11}(\pi_1) \quad (10a) \qquad\qquad \phi_2 = f_{21}(\pi_1) \quad (10b)$$

which relate ϕ_1, and ϕ_2, respectively, to π_1. The number of tasks that need to be rescheduled is

directly proportional to the number of processors that fail: N^{resch} increases as more processors fail. Thus, relationship $(10a)$ becomes as $N^{resch} = f_{11}(\mathbf{F})$. In particular,

$$N^{resch}(\mathbf{F}) = N_p^{resch}(\mathbf{F}) + N_{lb}^{resch}(\mathbf{F}) \quad (11)$$

where $N_p^{resch}(\mathbf{F})$ is the total number of tasks assigned to the failed resources that need to be rescued (or restarted), and $N_{lb}^{resch}(\mathbf{F})$ is the total number of 'surviving' tasks, assigned to 'surviving' resources, which the failure-recovery mechanism will need to reschedule together with $N_p^{resch}(\mathbf{F})$ with the goal of achieving and then maintaining a good load balancing on the remaining active processors. Additionally, $N_{lb}^{resch}(\mathbf{F})$ also depends on the choice of the DLS algorithm in use. It follows that the total parallel time T_{PAR} increases when the computational resources start to fail. Hence, T_{PAR} is expected to vary with respect to $\mathbf{F}$ and relationship $(10b)$ can be written as $T_{PAR} = f_{21}(\mathbf{F})$. The exact impact of $\mathbf{F}$ over T_{PAR} depends on the choice of DLS algorithm, as well as on its fault-discovery and fault-recovery mechanisms.

Step 4. At this step, we need to define the boundary values of $\mathbf{F}$, for which each element in Φ is less than the maximum tolerable number. Recall our discussion in the previous subsection on the role of considering the perturbation parameter as a discrete or a continuous variable. We consider $\mathbf{F}$ a *discrete* variable, that measures the number of "living" resources. In this case, we need to determine all the pairs of $\mathbf{F}$, such that for a given pair, the boundary value is the one that falls in the robust region. Assume that $\mathbf{F}'$ is a perturbation parameter value, such that the resources that fail in the scenario represented by $\mathbf{F}'$, include the resources that fail in the scenario represented by $\mathbf{F}$ and *exactly one* other resource. Then, the boundary relationships can be written as follows, in which T_{PAR}^{orig} is the estimated parallel time assuming that the system is completely safe, i.e.,

$\mathbf{F}^{orig} = [0\ 0 \ldots 0]^T$.

$$\left\{\mathbf{F} | \left(N^{resch}(\mathbf{F}) \leq \tau_2 N\right) \wedge \left(\exists \mathbf{F}' s.t.\ N^{resch}(\mathbf{F}') > \tau_2 N\right)\right\} \quad (12)$$

$$\left\{\mathbf{F} | \left(T_{PAR}(\mathbf{F}) \leq \tau_3 T_{PAR}^{orig}\right) \wedge \left(\exists \mathbf{F}' s.t.\ T_{PAR}(\mathbf{F}') > \tau_3 T_{PAR}^{orig}\right)\right\} \quad (13)$$

We define a robustness radius for this case in a similar manner to the previous case. Given the nature of the perturbation parameter, we use the ℓ_1-norm for determining the robustness radii:

$$r_{DLS}(N^{resch}, \mathbf{F}) = \max\ \|\mathbf{F} - \mathbf{F}^{orig}\|_1\ s.t.\ (N^{resch}(\mathbf{F}) \leq \tau_2 N)$$

$$\wedge(\exists \mathbf{F}' s.t.\ N^{resch}(\mathbf{F}') > \tau_2 N) \quad (14)$$

$$r_{DLS}(T_{PAR}, \mathbf{F}) = \max\ \|\mathbf{F} - \mathbf{F}^{orig}\|_1\ s.t.\ (T_{PAR}(\mathbf{F}) \leq \tau_3 T_{PAR}^{orig})$$

$$\wedge(\exists \mathbf{F}' s.t.\ T_{PAR}(\mathbf{F}') > \tau_3 T_{PAR}^{orig}) \quad (15)$$

$r_{DLS}(N^{resch}, \mathbf{F})$ is the *largest* number of resources that can fail in any combination without causing the N^{resch} to violate its tolerable value. Similarly, $r_{DLS}(T_{PAR}, \mathbf{F})$ is the *largest* number of resources that can fail in any combination without causing the degradation of T_{PAR} beyond $\tau_3 T_{PAR}^{orig}$. The overhead incurred by the fault-discovery and fault-recovery mechanisms of the DLS algorithm should be very small, such that the performance of the fault-tolerant DLS algorithm in the presence of failure achieves speedup over the sequential application. Moreover, assuming that the system is completely safe and no resource failures occur, T_{PAR}^{ft} given by the fault-tolerant DLS algorithm should be comparable to the T_{PAR} of the original non fault-tolerant DLS algorithm.

Finally, the robustness metric is the minimum of all robustness radii:

$$\rho_{DLS}(\Phi, \mathbf{F}) = \min\left(r_{DLS}(\phi_j, \mathbf{F})\right)\ \forall\ \phi_i \in \Phi \quad (16)$$

The *robustness metric* indicates that if the difference between the value of the actual number of "living" resources and the value of the resources assumed to be "living" is no larger than

$\rho_{DLS}(\Phi, \mathbf{F})$ percents (%), then N^{resch} will be at most τ_2 % of the total number of tasks, or that the estimated T_{PAR} time will be at most τ_3 times its original value. For our analysis, $\rho_{DLS}(\Phi, \mathbf{F})$ is the robustness metric of the "DLS" algorithm with respect to the number of tasks that have to be rescheduled, and the total parallel execution time, against resource failures.

This robustness metric can be used to determine the impact of resource failures on the performance of the three DLS, and differentiate them according to their reliability against such failures. Similar to the previous flexibility analysis, we distinguish between two possible scenarios, as illustrated in Figure 9. The first scenario, describes the situation when the three DLS methods perform similarly in terms of performance, with the difference being in the largest number of failures that they can handle and still achieve this performance. In this case, one should choose the DLS method that has the lowest impact on DLS performance and can handle the largest number of failures $\mathbf{F}$, in this case being AWF. The second scenario, depicts the case in which assuming that each DLS method is reliable agains the same number of resource failures, they yield different computational performance. Using our reliability metric, one is recommended AWF as the DLS of choice, for it has the lowest impact on DLS performance, regardless of the variation interval of $\mathbf{F}$.

7. Notes on the usefulness of the proposed metrics

The flexibility and reliability robustness metrics depend on certain application, system or algorithm specific parameters, most of which can be determined *apriori*. Hence, they can be formulated offline and integrated in the master to guide the autonomous scheduling process. If certain parameters become available (or known) only at runtime, the metrics are formulated using initial values

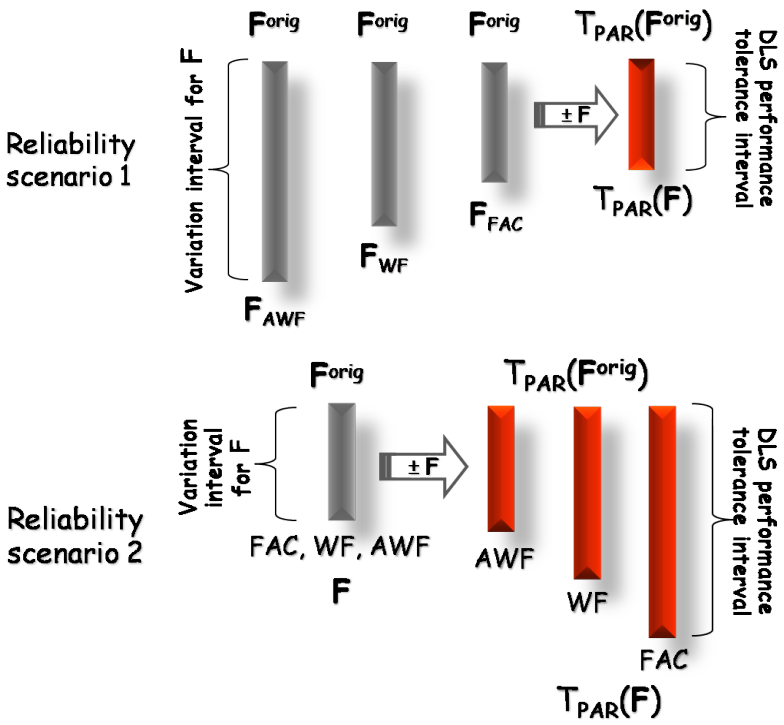

Figure 9. Possible scenarios to determine the reliability of FAC, WF and AWF

(e.g., every element of vector $\mathbf{F}$ is zero, meaning no failed resources), which are updated in the master when newer values become available (e.g., certain resources have failed, hence vector $\mathbf{F}$ contains non-zero elements). The choice of tolerance factors, τ_1, τ_2 and τ_3 can increase or decrease the robustness of DLS algorithms. In order to leverage the usefulness the proposed metrics, it is strongly recommended that the tolerance factors be chosen such that they reflect reality with high accuracy, rather than being selected based on personal choice. In Figure 10 we suggest some values for these parameters. Therefore, The usefulness of proposed metrics is twofold:

- the metrics can be formulated offline with application-, system-, and/or algorithm-specific initial values and integrated into the master to guide and adapt autonomously the scheduling

decisions

- usable in conjunction with other desired performance metrics (e.g. makespan) for differentiating among DLS that have similar performance from the makespan-only viewpoint

Tolerance factor	Depends on / Determined by	Suggested value
T_1	application type	1.2
T_2	DLS method of choice	50% of N
T_3	DLS method of choice, # of failures and fault detection & recovery mechanisms	$T_{PAR} \leq \tau_3 \cdot T^{orig}_{PAR} \wedge T_{PAR} < T^{fastest}_{SEQ}$

Figure 10. Suggested values for the tolerance factors

The choice of τ_1 depends on the nature of the application and it should be smaller (for instance $<$ 1.2 as suggested in 6.2) for real-time applications. Accordingly, it can be larger, if the application output is a single value (a cost or a yes/no answer), if the user might be interested in obtaining the result, while willing to wait a little longer for the output. The robustness of DLS algorithms under system loads variation can be improved based on the tolerable differences in the execution ratios reported to the manager by each group of processors. For a specific tolerable difference, we can maximize the robustness of the DLS algorithms under system load variations, by maximizing the tolerable increase in system load that would not violate the difference in the reported ratios. The key to maximally robust DLS algorithms under system load perturbations is to introduce a high degree of adaptivity in the scheduling decisions of the manager.

The second tolerance factor, τ_2, should be chosen such that the number of tasks that need to be rescheduled is not more than half of the total number of tasks, meaning $\tau_2 = 50\%$. An improvement

for this metric would be to express the tolerable impact of resource failures on the number of tasks to be rescheduled, such that it does not exceed the remaining number of tasks nor violate the boundary relationship for the total parallel time (eq. (13)), hence eliminating the need for τ_2.

The third factor τ_3 could be chosen such that for a given expected number of resource failures, the boundary conditions from eq. (13) are not violated, and in addition $\tau_3 T_{PAR}^{orig}$ does not exceed the sequential execution time of the application on the fastest processor in the platform. The key to maximally robust DLS algorithms in the presence of resource failures is devising a highly adaptable fault-tolerant mechanism that will handle failures (of processors, links, etc.) most efficiently. The existence of direct communication links between workers (of the same group and not only) provides a critical support for devising such fault-tolerant mechanisms (by enabling checkpointing, sharing of partial results, etc).

8. Computational cost analysis

The computational cost of each metric is driven by the computational cost of calculating each robustness radius. The problem of calculating a single robustness radius is equivalent to solving an optimization problem, and therefore it can generally be formulated as follows:

GENERAL OPTIMIZATION PROBLEM:

$$\textit{maximize } \sum_k ||\pi_k - \pi_k^{orig}||_p,\ k > 0$$

$$\textit{subject to } \beta_{min} \leq \{\Phi = f(\pi_k)\} \leq \beta_{max}$$

where $||\pi_k||_p$ is the L_p-norm of the perturbation parameter $\pi_k \in \Pi$, the interval $\langle \beta_{max}, \beta_{max} \rangle$ is the tolerance interval, $\Phi = f(\pi_k)$ is equivalent to $\Phi(\pi_k) = \tau \cdot \Phi(\pi_k^{orig})$, and τ is the tolerance

factor for the performance feature $\phi_k \in \Phi$ of interest.

More specifically, when the impact on performance feature of interest is due to variations in the system load, we can formulate the problem of calculating the robustness radius $r_{DLS}(T_{PAR}, \Lambda)$ as follows:

OPTIMIZATION PROBLEM FOR FLEXIBILITY:

$$\textit{maximize } \sum_k ||\Lambda - \Lambda^{orig}||_1,$$

$$\textit{subject to } {}_{PAR}(\Lambda) \leq \{\tau_1 \cdot_{PAR} (\Lambda^{orig)\}T_{PAR}(\Lambda) \leq_{SEQ}^{fastest}$$

Similarly, when the impact on performance feature of interest is due to resource failures, we formulate the problem of calculating the robustness radius $r_{DLS}(T_{PAR}, \mathbf{F})$ as follows:

OPTIMIZATION PROBLEM FOR RELIABILITY:

$$\textit{maximize } \sum_k ||\mathbf{F} - \mathbf{F}||_1,$$

$$\textit{subject to } N^{resch}(\mathbf{F}) \leq \tau_2 \cdot N_{PAR}(\mathbf{F}) \leq \{\tau_3 \cdot_{PAR} (\mathbf{F}^{orig)\}T_{PAR}(\mathbf{F}) \leq_{SEQ}^{fastest}$$

It is well known that all norms are convex function, hence the L_1-norm used above is a convex function. For a complete analysis we need to determine the nature of the function reflecting the impact of the perturbation parameters over the performance feature of interest, i.e. $\Phi(\Pi)$. Therefore $\Phi(\Lambda)$ and $\Phi((F))$ are the factors dominating the computational cost, and they can be linear, convex or concave functions. When both $\Phi(\Lambda)$ and $\Phi((F))$ are linear or convex functions, then the computational cost of determining the flexibility or reliability of a DLS method is equivalent to solving a convex optimization problem, for which inexpensive optimal solutions can be found. In the case when $\Phi(\Lambda)$ and $\Phi((F))$ are concave functions, the optimization problem is more complex and near-optimal solutions can be found with acceptable costs.

The metrics described in this report are the first step towards a robustness theory for autonomous complex workflow scheduling methods for high-performance multiscale numerical methods. These metrics have no effect when no perturbations occur in the parameters against which they quantify the robustness of a DLS algorithm. However, they offer valuable information for making scheduling decisions when perturbations do occur in those particular parameters, leading to *feasible*, *qualitative* and *efficient* schedules.

9. Conclusions and directions for future work

Scheduling today's applications, such as complex workflows that characterize multiscale numerical methods, on the latest computing platforms, such as cyberinfrastructure's high-performance resources, is very challenging, and among other attributes, must be realistic, efficient and robust. The metrics proposed in this work, combined with dynamic hierarchical management, are essential to bringing the most adaptive and efficient DLS algorithms to the state-of-the-art performance and robustness levels imposed by today's computing platforms and applications. Careful choice of the tolerance factors and incorporation into the hierarchical DLS methods, renders the proposed metrics useful towards producing efficient, autonomous, qualitative and reliable schedules for execution of large and complex scientific workflows

Directions for immediate and future work include: (1) devising robustness metrics for the adaptive DLS methods (adaptive factoring and recent variants of AWF), that use probabilistic analyses to model uncertainties in the execution environment; (2) using multiple performance parameters to devise realistic robustness metrics that give proper weight to their impact over each performance

features of interest; and (3) implementing these metrics and using them as performance metrics for evaluating the performance and execution robustness of the adaptive DLS methods in realistic uncertain large-scale high-performance computing platforms, individually or in combination to traditional performance metrics, such as makespan, resource utilization, etc. In order test and ensure the correct run-time behavior of complex computational workflows, the robustness metrics aid in providing a certain level of confidence about the completion and correct behavior of these workflows on the remote high-performance computational resources.

Acknowledgements

The funding provided for this study by the US Department of Energy under Grant No. 008860-013 is gratefully acknowledged. The distribution of this report is provided by the Center for Advanced Vehicular Systems of the Mississippi State University. This support is also appreciated.

References

[1] Gribble, S.D.: Robustness in Complex Systems. 8th Workshop on Hot Topics in Operating Systems (2001)

[2] T. Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. IEEE Trans. on Soft. Eng., 725–730 (1991)

[3] Hummel, S. F., Schonberg, E., Flynn, L.E.:Factoring: A Method for Scheduling Parallel Loops. Comm. of the ACM. 35:8, 90–101 (1992)

[4] Banicescu, I., Hummel, S. F.: Balancing processor loads and exploiting data locality in n-body simulations. Procs. of Supercomputing 95 (1995)

[5] Hummel, S.F., Schmidt, J., Uma, R.N., Wein, J.: Load-Sharing in Heterogeneous Systems via Weighted Factoring. Procs. SPAA, 318–328 (1996)

[6] Hurson, A., Lim, J., Kavi, K., Lee, B.: Parallelization of DOALL and DOACROSS Loops: A Survey. Advances in Computers, 45 (1997)

[7] Inacio, C., Ying, S.: Dependable Embedded Systems. http://www.ece.cmu.edu/~koopman/des_s99/robustness/draft.html (1998)

[8] Banicescu, I., Liu, Z.: Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes. Procs. high-performance Computing Symp. 122–129 (2000)

[9] Banicescu, I., Velusamy, V.: Load Balancing Highly Irregular Computations with the Adaptive Factoring, Procs. IPDPS '02, 195 (2002)

[10] Banicescu, I., Velusamy, V.: Performance of Scheduling Scientific Applications with Adaptive Weighted Factoring. Procs. IPDPS '01, 84 (2001)

[11] Kephart, J.O. and Chess, D.M.: The Vision of Autonomic Computing. Computer, 36:1, 41–50 (2003)

[12] Banicescu, I., Velusamy, V., Devaprasad, J.: On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring. Journal of Cluster Computing, 6:3, 215–226 (2003)

[13] Cariño, R. L., Banicescu, I., Rauber, T., Ruenger, G.: Dynamic Loop Scheduling with Processor Groups. Procs. Int'l Conf. P.&D. Comp. Systems (PDCS 2004), pp. 78-84 (2004)

[14] Cariño, R. L., Banicescu, I.: A Framework for Statistical Analysis of Datasets on Heterogeneous Clusters. Int'l Conf. on Cluster Comp., 1–9 (2005)

[15] Cariño, R. L., Banicescu, I.: A Load Balancing Tool for Distributed Parallel Loops. Journal of Cluster Computing, 8:3, 313–321 (2005)

[16] Cariño, R. L., Banicescu, I.: A Dynamic Load Balancing Tool for One and Two Dimensional Parallel Loops. 5th Int'l Symp. on Parallel and Distributed Computing (ISPDC '06), 107–114 (2006)

[17] Riakiotakis, I., Ciorba, F. M., Andronikos, T., Papakonstantinou, G.: Self-Adapting Scheduling for Tasks with Dependencies in Stochastic Environments. Procs. Cluster Computing/HeteroPar '06 (2006)

[18] Ali, S., Siegel, H. J., Maciejewski, A. A.: Perspectives on Robust Resource Allocation for Heterogeneous Parallel and Distributed Systems. Chapter 4 of Handbook of Parallel Computing Models, Algorithms and Applications. MK Publishing (2008)

[19] BioControl Jena GmBH: Survey 'Robustness in Immune System Models'. http://www.nisis.risk-technologies.com/downloads.aspx (2008)

[20] Benoit, A., Rehn-Sonigo, V., Robert, Y.: Optimizing Latency and Reliability of Pipeline Workflow Applications. RR2008-12 (2008)

[21] Chtepen, M., Claeys, F.H.A., Dhoedt, B., De Turck, F., Demeester, P., Vanrolleghem, P.A.: Adaptive Task Checkpointing and Replication Toward Efficient Fault-Tolerant Grids. IEEE Trans. on Par. and Dist. Systems, 20:2, 180–190 (2009)

Towards the robustness of dynamic loop scheduling on large-scale heterogeneous distributed systems

Ioana Banicescu
Dept. of Computer Science & Engineering
and Center for Computational Sciences
Mississippi State University
Mississippi State, USA, Email: ioana@cse.msstate.edu

Florina M. Ciorba and Ricolindo L. Cariño
Center for Advanced Vehicular Systems
Mississippi State University
Mississippi State, USA
Email: {florina,rlc}@cavs.msstate.edu

Abstract

Dynamic loop scheduling (DLS) algorithms provide application-level load balancing of loop iterates, with the goal of maximizing application performance on the underlying system. These methods use run-time information regarding the performance of the application's execution (for which irregularities change over time). Many DLS methods are based on probabilistic analyses, and therefore account for unpredictable variations of application and system related parameters. Scheduling scientific and engineering applications in large-scale distributed systems (possibly shared with other users) makes the problem of DLS even more challenging. Moreover, the chances of failure, such as processor or link failure, are high in such large-scale systems. In this paper, we employ the hierarchical approach for three DLS methods, and propose metrics for quantifying their robustness with respect to variations of two parameters (load and processor failures), for scheduling irregular applications in large-scale heterogeneous distributed systems.

1. Introduction

Researchers and scientists from various fields are interested in the accurate modeling and simulation of various complex phenomena from various scientific areas. These simulations are often routines that perform repetitive computations (in the form of DO/FOR loops) over very large data sets, and the number of repetitive computations (iterations) in these codes is not always constant. Moreover, their nature (or computational requirements) may be irregular, making one iteration likely to take more time than others, depending on the simulation. The resources in a large-scale system are widely distributed and highly heterogeneous, and as such, are usually shared among multiple users, and their availability cannot always be guaranteed or predicted. Hence, the *quality* and *quantity* of resources available to a single user changes continuously.

In this work, dynamic loop scheduling (DLS) techniques are considered to be the key solution for achieving and preserving the best performance of these applications in such environments. Herein, it is considered that a '*loop iteration*' (or a chunk of loop iterations) with variable execution time refers to a '*task*' (or a chunk of tasks, among many others within a loop of tasks) with variable execution time. A comprehensive description has been given earlier in a survey by Hurson et al. in [5] and in the relevant literature after that.

DLS methods provide two alternative approaches, *non-adaptive* and *adaptive*, for achieving good load balancing on variably loaded resources, as well as for executing tasks with varying execution times. Most of the techniques described in [5] are based on probabilistic analyses and are non-adaptive. Other non-adaptive techniques, which were not mentioned in the survey above, include fractiling [3] and weighted factoring [4]. Subsequent efforts gave birth to more elaborate techniques, called adaptive, and a few examples are given in [6][8][12]. Most of the above adaptive methods are based on probabilistic analyses, and use a combination of runtime information about the application and the system, in order to predict the system capabilities for the next computational assignments, or to estimate the time future tasks will require to finish execution, in order to achieve the best allocation possible for optimizing application performance via load balancing. In this paper, we employ a hierarchical management approach, and concentrate on two non-adaptive techniques, factoring [2] (FAC) and weighted factoring [4] (WF), and one adaptive technique, adaptive weighted factoring [7] (AWF). These techniques use probabilistic analyses to dynamically compute the size of chunks (a collection of tasks) at run-time, such that they are executed before their optimal time with high probability. Due to space limitations, the interested reader is referred to the appropriate references for details of the above DLS algorithms.

The performance of DLS methods using hierarchical management has been shown to be better than that of the centralized management approach [9][10][11]. Figure 1 illustrates the centralized management approach (left), and the distributed management approach (right). The coordination of and interactions between the processors in the first case are straightforward, whereas for details of the hierarchical management approach, due to space limitations, the interested reader if referred to [9].

Motivation Scheduling applications on large-scale plat-

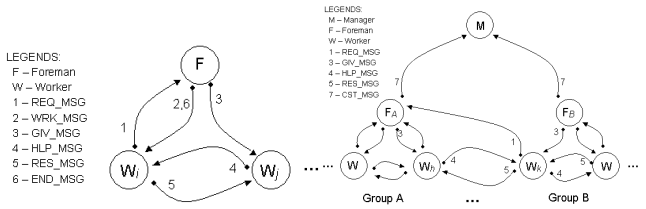

Figure 1. Left: centralized management, right: hierarchical management system

forms, where chances of faults are high, require an approach based on hierarchical management and mechanisms to ensure the robustness of the DLS methods. For this reason we consider FAC, WF and AWF, which are inherently robust because their design enables them to address unpredictabilities in the application and the system. Previously, the robustness of resource allocations/task scheduling algorithms was addressed individually for a single method, or even for a single application.

Contribution Inspired by the results in [9] we propose to employ the hierarchical approach in the DLS methods above. As such, the processors are organized into dynamically selected groups, such that the physical structure of the underlying platform is easily and well captured. The main contribution of our work lies in using the methodology proposed by Ali et al. [13] to propose two *metrics* that quantify the robustness of the hierarchical FAC, WF and AWF DLS algorithms, used for scheduling a very important *class* of applications: *irregular* tasks, against variations of two system related parameters: *load* and *processor failures*. The proposed metrics in conjunction with the hierarchical DLS methods provide quantitative and qualitative information such as: level of performance, quality of execution, for irregular applications in uncertain large-scale heterogeneous systems. These metrics alone are not more useful or better than any other performance measurement metrics (e.g. makespan, communication cost, resource utilization). Therefore, their use is mandatory towards achieving robustness for hierarchical DLS algorithms running on such uncertain systems.

The paper is organized as follows. Section 2 shows how to design robustness metrics and describes the two proposed metrics. Details regarding the implementation of these metrics and their usefulness are outlined in section 3. The paper is concluded in section 4 with an outline of directions for future work.

2. Robustness metrics design

Robustness is an emergent multifaceted phenomenon in advanced computing systems.

Designing specifically for robustness, to its full extent, is however not yet possible. Robustness benchmarks (or metrics) can be useful to measure *how* a scheduling method reacts to possible erroneous inputs or environmental factors, and hence, should employ mechanisms for *detecting* and *identifying* any erroneous parameters. In this work, we use the FePIA (features-perturbation-impact-analysis) procedure (see [13] and references therein for details), to design metrics that *model* and *estimate* the robustness of hierarchical DLS algorithms on large-scale realistic platforms against two perturbation parameters. The FePIA procedure consists of four general steps:

S.1 *Identify the performance features.*

S.2 *Identify the perturbation parameters.*

S.3 *Identify & clarify the impact of perturbation parameters* (S.2) *on performance features* (S.1).

S.4 *Identify the analysis to determine the robustness.*

Assuming that the application tasks are assumed to be independent and irregular, the goal of the hierarchical DLS algorithms is scheduling these tasks onto the set of P processors (divided into disjoint groups) of the large-scale heterogeneous distributed system, while minimizing the total parallel execution time (or *makespan*) T_{PAR}. A minimum T_{PAR} is achieved via dynamic load balancing, using processor speeds (which in the case of AWF are periodically adapted) and hierarchical management. Each processor in a group executes a set of tasks (called chunk) at a time. Each task is executed in a non-preemptive fashion, i.e., no other tasks of higher priority will suspend it. The same holds for the execution of a chunk, or for all chunks during a single time-step. Table 1 summarizes the notations we used in the following sections. The performance features of interest: $\mathcal{ET}_j$, T_{PAR}, and N_{resch}, and they should be limited in variation under certain application, system, or environment related parameters perturbations. For hierarchical DLS, perturbation parameters include variations in: task execution times (due only to irregularities of application characteristics), *system load* (available computational power of the processors given their shared use by multiple processors), network latency (delays in the communication speed due to network congestion), and *processor availability* (due to processor failures). Commonly, *all* perturbation parameters vary over time and cannot be accurately predicted before execution. A *robust hierarchical DLS algorithm* must adapt to any variations in these perturbation parameters, and yield performance parameters that vary in a constrained manner. Designing robustness metrics that incorporate *all* these parameters is very challenging [13].

2.1. Robustness against perturbations in system load

Assuming *unknown system load variations*, the individual finishing time $\mathcal{ET}_j$ of processor m_j must be robust against them. According to eq. (1), this translates into the requirement that the total parallel execution time T_{PAR} be robust against such variations, as well. Assuming system load variations, the *actual* finishing time $\mathcal{ET}_j$ of processor m_j must be calculated considering the effects of errors in the estimation of the processor's load variation, and *must not* exceed $\tau_1(> 1)$ times its estimated value $\mathcal{ET}_j^{orig}$, where

N	total number of tasks
N^{resch}	# of tasks that need to be *rescheduled*
a_i	i-th task, $1 \leq i \leq N$
P	total number of processors
m_j	j-th processor, $1 \leq j \leq P$
T_{ij}	*execution* time of task a_i on m_j
T_{ij}^{W2F}	*communication* time between m_j and its foreman for executing a_i
T_{ij}^{W2W}	*communication* time between m_j and any other workers for executing a_i (processor regrouping)
$\mathcal{ET}_j$	*finishing* time of all tasks computed by m_j
T_{PAR}	total parallel execution time for the N tasks
$\lambda = [\lambda_1 \dots \lambda_P]^T$	vector of processors *load* (= system load)
$\mathbf{F} = [\mathbf{f}_1 \dots \mathbf{F}_P]^T$	vector of processors status (active/*failed*) vector
$\Phi = \{\phi_1, \dots\}$	set of *performance* features
$\Pi = \{\pi_1, \dots\}$	set of *perturbation* parameters
τ_1, τ_2, τ_3	tolerance factors for performance features
$r_{DLS}(,)$	robustness radius
$\rho_{DLS}(,)$	robustness metric

Table 1. Notation

τ_1 is a tolerance factor reflecting the robustness. The FePIA procedure for this analysis is outlined below.

S.1 Let $\Phi = \{\phi_1\}$, $\phi_1 = \mathcal{ET}_j$, $1 \leq j \leq P$ be the performance features set. The individual finishing time for all {tasks $i|a_i$ executed on m_j}, is:

$\mathcal{ET}_j = \sum_{i,j}^{N,P} \left(T_{ij} + T_{ij}^{W2F} + T_{ij}^{W2W}\right)$ (2)

S.2 Let the perturbation parameters set be $\Pi = \{\pi_1\}$, $\pi_1 = \lambda_j$, $1 \leq j \leq P$. We consider λ_j to be the *individual* load of processor m_j, and λ the vector that contains all processors load values. Initially, the DLS *assumes* that the system has λ^{orig} load, which can be usually determined by executing the first batch of chunks, as determined by the original factoring rules and their subsequent evolution. The j-th position in the λ^{orig} vector is the initial load of m_j.

S.3 The impact of λ_j over $\mathcal{ET}_j$, is determined by analyzing individually, for all processors, their finishing time given their *own* load. Each actual finishing time is expected to vary according to λ_j, denoted as $\mathcal{ET}_j(\lambda_j)$. Mathematically, for all {tasks $i|a_i$ executed on m_j under varying load λ_j}, this is written as:

$\mathcal{ET}_j(\lambda_j) = \sum_{i,j}^{N,P} \left(T_{ij}(\lambda_j) + T_{ij}^{W2F}(\lambda_j) + T_{ij}^{W2W}(\lambda_j)\right)$ (3)

S.4 We must define the boundary values of $\pi_1 = \lambda_j$. First, we must decide whether the perturbation parameter is a continuous or a discrete variable. There are two ways to measure the load of a processor in heterogeneous systems: number of processes in the processor's run-queue [1] (discrete variable), or delivered processor speed, measured as percentage of processor availability [**?**] (continuous variable). In this work, λ_j is a *continuous* variable measuring processor availability, which is highly advantageous since it expresses the delivered processor speed, which in fact reflects simultaneously the impact of: applications' requirements, hardware capabilities, and network speed in one. The boundary values of λ_j must satisfy the following boundary relationships:

$\left\{\lambda_j \in \left\langle \lambda_j', \lambda_j'' \right\rangle \mid \left(f_1(\lambda_j') = \beta_1^{\min}\right) \wedge \left(f_1(\lambda_j'') = \beta_1^{\max}\right)\right\}$ (4)

The tolerable variation interval for $\mathcal{ET}_j$ (the performance feature of interest), is given by $\left\langle \beta_1^{\min}, \beta_1^{\max} \right\rangle$. The tolerable increase in the *actual* finishing time $\mathcal{ET}_j$ of processor m_j, considering the effects of errors in the estimation of variations of λ_j, cannot exceed $\tau_1(>1)$ times its estimated value $\mathcal{ET}_j^{orig}$. The boundary relationships for this analysis are:

$\left\{\lambda_j \in \left\langle \lambda_j', \lambda_j'' \right\rangle \mid \left(\mathcal{ET}_j(\lambda_j) = \tau_1 \mathcal{ET}_j^{orig}\right) \wedge (1 \leq j \leq P)\right\}$ (5)

The robustness radius, $r_{DLS}(\mathcal{ET}_j, \lambda_j)$, is expressed as the *largest* increase in processor load, for any combination of processor load values, from the assumed value, that does *not* cause any tolerance interval violation for the execution time of all tasks a_i assigned to m_j. We must choose the norm which yield the smallest variation in the system (and ultimately processor) load, and we believe that a more intuitive norm to use is the ℓ_1-norm, and $r_{DLS}(\mathcal{ET}_j, \lambda_j)$ can be written as follows:

$r_{DLS}(\mathcal{ET}_j, \lambda_j) = \min \ \|\lambda_j - \lambda_j^{orig}\|_1 \ s.t. \ \mathcal{ET}_j(\lambda_j) = \tau_1 \mathcal{ET}_j^{orig}$ (6)

The *robustness metric* is the minimum of all robustness radii: $\rho_{DLS}(\Phi, \lambda_j) = \min\left(r_{DLS}(\phi_1, \lambda_j)\right) \ \forall \ \phi_i \in \Phi$ (7)

An acceptable value for τ_1 was proposed in [13] to be 1,2. For this analysis, $\rho_{DLS}(\mathcal{ET}_j, \lambda_j)$ is the general robustness metric of the "DLS" algorithm, with respect to each processor's individual finishing execution time against perturbations in the processor load, and $\rho_{DLS}(T_{PAR}, \lambda)$ is the robustness metric of T_{PAR} against variations in the total system load:

$\rho_{DLS}(T_{PAR}, \lambda) = \min(\rho_{DLS}(\mathcal{ET}_j, \lambda_j)), 1 \leq j \leq P$ (8).

2.2. Robustness against processor failures

Assuming an un-safe system with expected failures, N^{resch} and T_{PAR}, must both be robust against them: N^{resch} *must not* exceed τ_2% of the total number of tasks N, and T_{PAR} *must not* exceed $\tau_3(>1)$ times it's estimated value T_{PAR}^{orig} (computed under the assumption that the system is completely safe).

When failures occur, the DLS algorithm must be able to reschedule the tasks that were assigned to the failed processors, as well as other tasks if necessary (for instance, to preserve load balancing on the remaining processors). To reduce the complexity of the analysis, we make the following simplifying assumptions: (i) only worker processors fail, (ii) failures occur simultaneously, and (iii) failures are permanent. We also assume that the DLS algorithm has fault-discovery and fault-recovery mechanisms. These assumptions can be relaxed in order to deliver more optimal and general robustness metrics. The FePIA procedure for this analysis is given below.

S.1 In this case the set of performance features has two elements: $\Phi = \{\phi_1, \phi_2\}$, $\phi_1 = N^{resch}$ and $\phi_2 = T_{PAR}$.

S.2 To identify the failing processors, we use consider $\mathbf{F} = [\mathbf{f}_1 \mathbf{f}_2 \dots \mathbf{f}_P]^T$ as the vector containing the statuses of all processors, defined as $\mathbf{f}_j = 1$ if processor m_j failed, and $\mathbf{f}_j = 0$ otherwise, $1 \leq j \leq P$.

$\mathbf{F}^{orig} = [0 \ 0 \dots 0]^T$, indicates that all processors are initially active. The perturbation parameters set is $\Pi = \{\pi_1 = \mathbf{F}\}$.

S.3 To determine the impact of Π over Φ, we need to determine separately each of the following:

$\phi_1 = f_{11}(\pi_1)$ (9a) $\phi_2 = f_{21}(\pi_1)$ (9b)

which relate ϕ_1, and ϕ_2, respectively, to π_1. N^{resch} is directly proportional to the number of failing processors. Thus, (9a) becomes $N^{resch}(\mathbf{F}) = N_p^{resch}(\mathbf{F}) + N_{lb}^{resch}(\mathbf{F})$ (10) where $N_p^{resch}(\mathbf{F})$ is the total number of tasks assigned to the processors that failed, and $N_{lb}^{resch}(\mathbf{F})$ is the total number of tasks that need to be rescheduled in order to achieve and maintain a good load balancing on the remaining active processors. Additionally, $N_{lb}^{resch}(\mathbf{F})$ depends also on the choice of the DLS algorithm in use. T_{PAR} increases when processors start to fail. Hence, relationship (9b) becomes $T_{PAR} = f_{21}(\mathbf{F})$. The exact impact of $\mathbf{F}$ over T_{PAR} depends on the choice of DLS algorithm and its fault-recovery mechanism.

S.4 To define the boundary values of $\pi_1 = \mathbf{F}$ for each element in Φ, we consider $\mathbf{F}$ a *discrete* variable that measures the number of "living" processors. We need to determine all the pairs of $\mathbf{F}$, such that for a given pair, the boundary value is the one that falls in the robustness region. Assume that $\mathbf{F}'$ is a perturbation parameter value, such that the machines that fail in the scenario represented by $\mathbf{F}'$ include the machines that fail in the scenario represented by $\mathbf{F}$ and *exactly one* other machine. Then, the boundary relationships are: $\left\{\mathbf{F} | \left(N^{resch}(\mathbf{F}) \leq \tau_2 N\right) \wedge \left(\exists \mathbf{F}' s.t.\ N^{resch}(\mathbf{F}') > \tau_2 N\right)\right\}$ (11)
$\left\{\mathbf{F} | \left(T_{PAR}(\mathbf{F}) \leq \tau_3 T_{PAR}^{orig}\right) \wedge \left(\exists \mathbf{F}' s.t.\ T_{PAR}(\mathbf{F}') > \tau_3 T_{PAR}^{orig}\right)\right\}$ (12)

where T_{PAR}^{orig} is the estimated parallel time assuming that the system is completely safe. We define the robustness radii for this case using the ℓ_1-norm: $r_{DLS}(N^{resch}, \mathbf{F}) = \min \|\mathbf{F} - \mathbf{F}^{orig}\|_1\ s.t.\ (N^{resch}(\mathbf{F}) \leq \tau_2 N)$
$\wedge (\exists \mathbf{F}' s.t.\ N^{resch}(\mathbf{F}') > \tau_2 N)$(13)
$r_{DLS}(T_{PAR}, \mathbf{F}) = \min \|\mathbf{F} - \mathbf{F}^{orig}\|_1\ s.t.\ (T_{PAR}(\mathbf{F}) \leq \tau_3 T_{PAR}^{orig})$
$\wedge (\exists \mathbf{F}' s.t.\ T_{PAR}(\mathbf{F}') > \tau_3 T_{PAR}^{orig})$(14)

$\rho_{DLS}(\Phi, \mathbf{F})$ is the robustness metric of the "DLS" algorithm against processor failures, with respect to N^{resch}, and T_{PAR}: $\rho_{DLS}(\Phi, \mathbf{F}) = \min\left(r_{DLS}(\phi_j, \mathbf{F})\right)\ \forall\ \phi_i \in \Phi$ (15)

3. Implementation and usefulness

An analysis of the computational complexity for computing such metrics based on the FePIA procedure is given in [13] (and references therein). The choice of τ_1, τ_2 and τ_3 impacts the robustness of DLS algorithms, and the proposed metrics are useful if these factors reflect reality with high accuracy. The metrics depend on certain application, system or algorithm specific parameters, most of which can be determined *apriori*. Hence, the metrics can be formulated offline and injected in the master to guide the dynamic scheduling process. If certain parameters become available (or known) only at runtime, the metrics are formulated using initial values (e.g., every element of vector $\mathbf{F}$ is zero, meaning no failed processors), which are updated in the master when newer values become available (e.g., certain processors failed, hence vector $\mathbf{F}$ contains non-zero elements). These metrics have no effect when no perturbations occur in the parameters against which they quantify the robustness of a DLS algorithm. However, they offer valuable information for making scheduling decisions when perturbations do occur in those particular parameters, leading to *feasible*, *qualitative* and *efficient* schedules.

4. Conclusions and future work

Scheduling today's applications on the latest computing platforms is challenging, and among other attributes, it must be realistic, efficient and robust. The metrics proposed in this work, in combination with the dynamic hierarchical management approach, are essential to bringing the most adaptive and efficient DLS algorithms to the state-of-the-art level required by today's computing platforms and applications. Immediate and future work directions include: devising similar robustness metrics for the adaptive DLS methods (adaptive factoring and recent variants of AWF), that use probabilistic analyses to model uncertainties; using multiple performance parameters and devise realistic robustness metrics that give proper weight to their impact over each performance features of interest; implementing these metrics and using them as performance metrics for evaluating the adaptive DLS methods in realistic large-scale platforms, individually against or in combination to traditional performance metrics, such as makespan, resource utilization, etc.

References

[1] T. Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. IEEE Trans. on Soft. Eng., 725–730 (1991)

[2] Hummel, S. F., Schonberg, E., Flynn, L.E.:Factoring: A Method for Scheduling Parallel Loops. Comm. of the ACM. 35:8, 90–101 (1992)

[3] Banicescu, I., Hummel, S. F.: Balancing processor loads and exploiting data locality in n-body simulations. Procs. of Supercomputing 95 (1995)

[4] Hummel, S.F., Schmidt, J., Uma, R.N., Wein, J.: Load-Sharing in Heterogeneous Systems via Weighted Factoring. Procs. SPAA, 318–328 (1996)

[5] Hurson, A., Lim, J., Kavi, K., Lee, B.: Parallelization of DOALL and DOACROSS Loops: A Survey. Advances in Computers, 45 (1997)

[6] Banicescu, I., Velusamy, V.: Load Balancing Highly Irregular Computations with the Adaptive Factoring, Procs. IPDPS '02, 195 (2002)

[7] Banicescu, I., Velusamy, V.: Performance of Scheduling Scientific Applications with Adaptive Weighted Factoring. Procs. IPDPS '01, 84 (2001)

[8] Banicescu, I., Velusamy, V., Devaprasad, J.: On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring. Journal of Cluster Computing, 6:3, 215–226 (2003)

[9] Cariño, R. L., Banicescu, I., Rauber, T., Ruenger, G.: Dynamic Loop Scheduling with Processor Groups. Procs. Int'l Conf. P.&D. Comp. Systems (PDCS 2004), pp. 78-84 (2004)

[10] Cariño, R. L., Banicescu, I.: A Framework for Statistical Analysis of Datasets on Heterogeneous Clusters. Int'l Conf. on Cluster Comp., 1–9 (2005)

[11] Cariño, R. L., Banicescu, I.: A Dynamic Load Balancing Tool for One and Two Dimensional Parallel Loops. 5th Int'l Symp. on Parallel and Distributed Computing (ISPDC '06), 107–114 (2006)

[12] Riakiotakis, I., Ciorba, F. M., Andronikos, T., Papakonstantinou, G.: Self-Adapting Scheduling for Tasks with Dependencies in Stochastic Environments. Procs. Cluster Computing/HeteroPar '06 (2006)

[13] Ali, S., Siegel, H. J., Maciejewski, A. A.: Perspectives on Robust Resource Allocation for Heterogeneous Parallel and Distributed Systems. Chapter 4 of Handbook of Parallel Computing Models, Algorithms and Applications. MK Publishing (2008)