Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 68 (2008) 1157–1173
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

# Stochastic robustness metric and its use for static resource allocations

Vladimir Shestak[a,c], Jay Smith[a,d], Anthony A. Maciejewski[a], Howard Jay Siegel[a,b,*]

[a] *Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO 80523, United States*
[b] *Department of Computer Science, Colorado State University, Fort Collins, CO 80523, United States*
[c] *InfoPrint Solutions Company 6300 Diagonal Highway Boulder, CO 80301, United States*
[d] *Digital Globe 1601 DryCreek Rd Longmont, CO 80503, United States*

Received 20 June 2007; received in revised form 22 December 2007; accepted 4 January 2008
Available online 10 March 2008

**Abstract**

This research investigates the problem of robust static resource allocation for distributed computing systems operating under imposed Quality of Service (QoS) constraints. Often, such systems are expected to function in a physical environment replete with uncertainty, which causes the amount of processing required to fluctuate substantially over time. Determining a resource allocation that accounts for this uncertainty in a way that can provide a probabilistic guarantee that a given level of QoS is achieved is an important research problem. The stochastic robustness metric proposed in this research is based on a mathematical model where the relationship between uncertainty in system parameters and its impact on system performance are described stochastically.

The utility of the established metric is then exploited in the design of optimization techniques based on greedy and iterative approaches that address the problem of resource allocation in a large class of distributed systems operating on periodically updated data sets. The performance results are presented for a simulated environment that replicates a heterogeneous cluster-based radar data processing center. A mathematical performance lower bound is presented for comparison analysis of the heuristic results. The lower bound is derived based on a relaxation of the Integer Linear Programming formulation for a given resource allocation problem.


*Keywords:* Heterogeneous systems; Stochastic resource allocation; Heuristic methods; QoS; Constrained optimization; Distributed systems; Robustness

## 1. Introduction

Often, parallel and distributed computing systems must operate in an environment replete with uncertainty while providing a required level of quality of service (QoS). This reality has inspired an increasing interest in robust design. The following are some examples. The Robust Network Infrastructures Group at the Computer Science and Artificial Intelligence Laboratory at MIT takes the position that "... a key challenge is to ensure that the network can be robust in the face of failures, time-varying load, and various errors". The research at the User-Centered Robust Mobile Computing Project at Stanford "concerns the hardening of the network and software infrastructure to make it highly robust". The Workshop on Large-Scale Engineering Networks: Robustness, Verifiability, and Convergence (2002) concluded that the "Issues are... being able to quantify and design for robustness...". There are many other projects of similar nature at other universities and organizations.

To provide insight into the target systems operating under uncertainty that must maintain a certain level of QoS, consider the following two examples. Fig. 1 schematically depicts part of a total ship computing environment in the Adaptive and Reflective Middleware Systems (ARMS) program supported by DARPA's Information Exploitation Office [4]. This part of the ARMS example represents a large class of systems that operate on *periodically* updated data sets, e.g., defense surveillance for homeland security, and monitoring vital signs of medical patients. Typically, in such systems, sensors (e.g., radar, sonar, and video camera) produce data sets with a constant period of $\Lambda$ time units. Periodic data updates imply that the total processing time for any given data set must not exceed $\Lambda$, i.e., $\Lambda$ is an

* Corresponding author at: Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO 80523, United States.
*E-mail addresses:* shestak@colostate.edu (V. Shestak), jtsmith@digitalglobe.com (J. Smith), aam@colostate.edu (A.A. Maciejewski), hj@colostate.edu (H.J. Siegel).

0743-7315/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.01.002

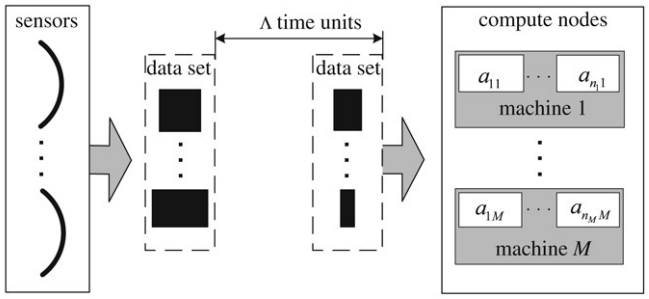


Fig. 1. The ARMS example: major functional units and data flow for a class of systems that operate on periodically updated data sets. The $a_{ij}$'s denote applications executing on machine $j$. Processing of each data set must be completed within Λ time units.

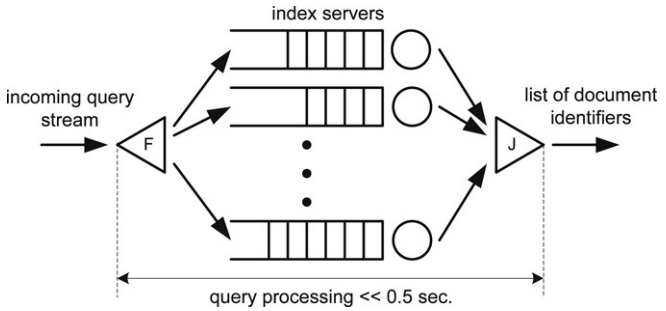


Fig. 2. The Google example: Fork (F) and Join (J) query processing executed by index servers in the first phase of the search engine.

imposed timing QoS constraint for the system. Suppose that each input data set must be processed by a collection of $N$ independent applications that can be executed in parallel on the available set of $M$ heterogeneous compute nodes. Due to the changing physical world, the periodic data sets produced by the system sensors typically vary in such parameters as the number of observed objects present in the radar scan and signal-to-noise ratio. Variability in the data sets results in variability in the execution times of processing applications. Due to an inability to precisely predict application execution times, they can be considered uncertainty parameters in the system.

Another example of a distributed computing system that must accommodate uncertainty under tight timing QoS constraints is a web search engine. In the Google search engine [5], the user query response time is required to be at most 0.5 s—including network round trip communication latency. Query execution in this system consists of two major phases. The first phase produces an ordered list of document identifiers. This list is a result of merging the responses from multiple index servers, each searching over a particular subset of the entire index database. The second phase uses the list of document identifiers and computes the actual title and uniform resource locators of these documents, along with any query-specific document summary information. Document servers perform this job, each processing a certain part of the list.

Consider the first phase of the system where a fork-join job [25] must be performed, as shown in Fig. 2 (similar analysis can be derived for the second phase). To reduce overall execution time, each query is duplicated and processed in parallel by a subset of the available index servers—chosen by the cluster manager such that they cover the entire index database. Each copy queues to a different index server, and each index server has its own input buffer where the requests are serviced in the order of their arrival (for simplicity of analysis, sequential query processing at each index server is considered in this study). The cluster manager must be able to accommodate uncertainty in query processing times because the exact time required to process a query is not known *a priori*. However, it is possible for the cluster manager to use the attributes of an incoming query to identify a subset of the past queries that have similar attributes and share a common distribution of execution times. These past execution times taken from the identified subset of queries can be used to create a probability density function (pdf) that describes the possible execution times for the incoming query.

According to [2], any claim of robustness for a given system must answer three questions: (a) what behavior of the system makes it robust? (b) what uncertainties is the system robust against? (c) quantitatively, exactly how robust is the system? As an example, consider the ARMS environment shown in Fig. 1, where the system is robust if it is capable of processing each data set within Λ time units. A resource allocation deployed in this system must be robust against uncertainties in execution times of the applications processing data sets. In our approach, the degree of robustness is measured as the probability that all of the processing required for a given data set is completed within Λ time units. Very similar definitions could be derived for the Google example.

In both examples, an important task for a resource management system is to distribute applications (or queries) across compute nodes (or index servers) such that the produced resource allocation is *robust*, i.e., it can guarantee (or has a high probability) that the imposed QoS constraint is satisfied despite uncertainties in processing times. Simple load balancing algorithms may be sufficient when a distributed system is not over-subscribed, i.e., the number of queued tasks at each compute node is small, so tasks can be completed well before their deadlines. However, more sophisticated stochastic analysis is required for resource allocations as the system experiences workload surges or a loss of resources. Robust design for such systems involves determining a resource allocation that can account for uncertainty in a way that enables the system to provide a probabilistic guarantee that a given QoS is achieved. Our study defines a stochastic methodology for quantifiably determining the ability of a resource allocation to satisfy QoS constraints in the midst of uncertainty in system parameters.

The problem of resource allocation in the field of heterogeneous parallel and distributed computing is NP-complete (e.g., [11,20]), therefore, the development of heuristic techniques to find near-optimal solutions represents a large body of research (e.g., [1,9,13,15,16,20,26,29,40]). There are two major classes of resource allocation approaches widely used in practice: greedy heuristics and iterative algorithms. Usually, greedy heuristics are relatively fast (as opposed to time-consuming global search heuristics), as they generate a solution by making locally optimal decisions; this feature

often makes greedy heuristics an appropriate choice to use in dynamic (i.e., on-line mapping) systems. However, the quality of solutions produced by greedy heuristics is generally lower than that produced by global search heuristics that progressively improve a solution through multiple iterations.

In the first part of this work, a new stochastic robustness metric is presented where the uncertainty in system parameters and its impact on system performance are modeled stochastically. This stochastic model is then used to derive a quantitative evaluation of the robustness of a given resource allocation as the probability that the resource allocation will satisfy the expressed QoS constraints. Two alternative means for computing the metric are presented that render the required computation practical in a number of common environments. The utility of the proposed stochastic metric is analyzed in a simulated environment by comparing it against existing deterministic metrics, i.e., metrics where outcomes are not associated with probabilities.

In the second part of this work, the proposed method of stochastic robustness evaluation was integrated into greedy and global search heuristics developed to address the problem of resource allocation for a class of distributed systems operating on periodic data sets schematically depicted in Fig. 1. In many systems of the considered class, it is highly desirable to *minimize* the period $\Lambda$ between subsequent data arrivals while providing a probabilistic guarantee that each data set is processed within $\Lambda$ time units. As a practical example, consider air traffic control and military applications where frequent radar scans are needed to identify an approaching target with a guaranteed high probability of successful processing of each scan.

In summary, the two major contributions of this work include: (1) the development of a mathematical model for a stochastic robustness metric that utilizes available information to quantifiably determine the ability of a resource allocation to satisfy expressed QoS constraints; and (2) the design and performance analysis of optimization techniques that solve the problem of robust resource allocation in distributed systems operating on periodically updated data sets. We demonstrate that when the distributions of random variables associated with uncertain parameters in the stochastic model are available, an evaluation of a resource allocation leads to more useful results than that achievable with deterministic metrics utilizing mean values. Major contributions of this work are a discussion on the applicability of FFT and the bootstrap method for computing the proposed stochastic robustness metric and the derivation of a lower bound on a minimum $\Lambda$ achievable based on our Integer Linear Programming relaxation. We also examine the literature pertinent to the area of robust resource allocation in distributed systems.

In Section 2, a formal definition of stochastic robustness is given, while Section 3 discusses methods of computing the stochastic robustness metric given the independence of input parameters. A comparison study demonstrating the effectiveness of the proposed robustness measure versus deterministic metrics is included in Section 4. The descriptions of the heuristics or generating a robust resource allocation that utilize the new metric are presented in Sections 5 and 6 for greedy and iterative approaches respectively. This is followed by a proof of an effective lower bound in Section 7, which is used for comparison in the performance analysis. Section 8 contains the details of the simulation setup. The performance results of the developed heuristics are presented in Section 9. A discussion of the relation of this study to the published work from the literature is given in Section 10. A glossary of notation and acronyms used in the paper are tabulated in the Appendix.

## 2. Mathematical model for stochastic robustness

A stochastic robustness metric for a given distributed computing environment should reasonably predict the performance of the system. Given the existing content in the ARMS example, let $S_j$ be the sequence of $n_j$ applications assigned to compute node $j$ in the order they are to be executed, i.e., $S_j = [a_{1j}, a_{2j}, ...., a_{n_j j}]$. In the Google example, the sequence $S_j$ represents $n_j$ queries assigned to index server $j$. Let random variable $T_{ij}$ denote the execution time of each individual application (or query) $a_{ij}$ on compute node (or index server) $j$. The random variables $T_{ij}$ characterize the uncertainty in execution time for each of the applications in the system and serve as the inputs to the mathematical model. These random variables are the uncertainty parameters in the mathematical model.

In the ARMS example, the evaluation of system performance is based on the makespan value (total time required for all applications to process a given data set) [9] achieved by a given resource allocation, i.e., a smaller makespan equates to better performance. The functional dependence between the uncertainty parameters and the performance characteristic, denoted as $\psi$, in the model is

$$\psi = \max_{j=1,\ldots,M} \left\{ \sum_{i=1}^{n_j} T_{ij} \right\}. \tag{1}$$

In the Google example, the performance in the first phase is measured for each individual query. Unlike the ARMS example, where the evaluation of makespan values occurs at each $\Lambda$, query performance evaluation in the Google example is performed while the system is busy processing queries. Assume that $M$ copies of a query arrive at index servers at wall-clock time $t$, and $n_j$ is the number of queries pending execution or being executed by index server $j$ at that time. Let $t_{0j}$ denote the wall-clock start time of execution for the query being processed by index server $j$ at time $t$. The functional dependence between the uncertainty parameters and the performance characteristic at time $t$, denoted as $\psi(t)$, is

$$\psi(t) = \max_{j=1,\ldots,M} \left\{ T_{1j} - (t - t_{0j}) + \sum_{i=2}^{n_j} T_{ij} \right\}. \tag{2}$$

Due to its functional dependence on the uncertainty parameters $T_{ij}$, the performance characteristic in Eqs. (1) and (2) is itself a random variable.

Let the QoS constraints be quantitatively described by the values $\beta_{\min}$ and $\beta_{\max}$ limiting the acceptable range of possible variation in system performance [2], i.e., $\beta_{\min} \leq \psi \leq$

$\beta_{\max}$. *The stochastic robustness metric, denoted as* $\theta$, *is the probability that the performance characteristic of the system is confined to the interval* $[\beta_{\min}, \beta_{\max}]$, *i.e.*, $\theta = \mathbb{P}[\beta_{\min} \leq \psi \leq \beta_{\max}]$. For a given resource allocation, the stochastic robustness quantitatively measures the probability that the generated system performance will satisfy the stipulated QoS constraints. Clearly, unity is the most desirable stochastic robustness metric value, i.e., there is zero probability that the system will violate the established QoS constraints.

## 3. Computational issues

### 3.1. Assumptions of independence

In the model of compute node $j$, the functional dependence between the set of local uncertainty parameters $\{T_{ij} | 1 \leq i \leq n_j\}$ and the local performance characteristic $\psi_j$ can be stated in the ARMS example as $\psi_j = \sum_{i=1}^{n_j} T_{ij}$; in the Google example as $\psi_j = T_{1j} - (t - t_{0j}) + \sum_{i=2}^{n_j} T_{ij}$.

Independence of the local performance characteristics implies that the random variables $\psi_1, \psi_2, \ldots, \psi_M$ are mutually independent. If such independence is established, the stochastic robustness in a distributed system can be expressed as the product of the probabilities of each compute node meeting the imposed QoS constraints. Mathematically, this is given as

$$\theta = \prod_{j=1}^{M} \mathbb{P}[\beta_{\min} \leq \psi_j \leq \beta_{\max}]. \tag{3}$$

Specifically in Eq. (3), $\beta_{\max} = \Lambda$ in the ARMS example and $\beta_{\max} \ll 0.5$ s in the Google example. In both examples, $\beta_{\min}$ is set to 0 because there is no minimum time constraint on execution.

If the execution times $T_{ij}$ of applications mapped on a compute node $j$ are mutually independent, then $\mathbb{P}[\beta_{\min} \leq \psi \leq \beta_{\max}]$ can be computed by taking the integral between $\beta_{\min}$ and $\beta_{\max}$ over the completion time pdf for machine $j$. The completion time distribution for machine $j$ is determined by taking an $n_j$-fold convolution of probability density functions (pdfs) $f_{T_{ij}}(t_i)$ [27]. The probability can then be computed as

$$\mathbb{P}[\beta_{\min} \leq \psi_j \leq \beta_{\max}] = \int_{\beta_{\min}}^{\beta_{\max}} [f_{T_{1j}}(t_1) * \cdots * f_{T_{n_j j}}(t_{n_j})] \mathrm{d}t. \tag{4}$$

This assumption of independence is valid for non-multitasking execution mode which is commonly considered in the literature [9,13,25,29,40], and applied in practice in a variety of systems, e.g., an iterative UDP server model [17].

### 3.2. Fast Fourier transform method

An $n_j$-fold convolution in Eq. (4) requires $n_j - 1$ computations of the convolution integral [27]; thus, a direct numerical integration may become a formidable task when $n_j$ is a relatively large number. However, a high quality approximation to the $n_j$-fold convolution can be obtained, at a low computational expense, by applying Fourier transforms. Thus, if $\Phi_{T_{ij}}(\omega)$ denotes the characteristic function of $T_{ij}$, i.e., the forward Fourier transform [33], and $\Phi_{\psi_j}^{-1}$ denotes the inverse Fourier transform, then Eq. (4) can be computed as follows

$$\mathbb{P}[\beta_{\min} \leq \psi_j \leq \beta_{\max}] = \int_{\beta_{\min}}^{\beta_{\max}} \Phi_{\psi_j}^{-1}\{\Phi_{T_{1j}}(\omega) \times \cdots \times \Phi_{T_{n_j j}}(\omega)\} \mathrm{d}t. \tag{5}$$

From this point onwards, assume that each pdf $f_{T_{ij}}(t_i)$ is expressed as a discrete probability mass function (pmf) utilizing $\Omega$ points—this is common in practical implementations. As such, the calculation can be performed using a Fast Fourier Transform method (FFT) that reduces the computational cost of finding the corresponding characteristic functions $\Phi_{T_{ij}}$. The FFT method is a discrete Fourier transform algorithm that reduces the number of computations needed for $\Omega$ points from $2\Omega^2$ to $2\Omega \log \Omega$ [33]. Thus, the computational complexity of determining the local performance characteristic can be drastically reduced, making the approach reasonable to compute.

In dynamic systems (i.e., on-line mapping), processing a continuous stream of tasks (e.g., in the Google example), the number of convolutions required at each mapping event is relatively low. For example, evaluating a potential allocation of a given task on a particular compute node requires only one convolution of the *execution* time distribution for the task with the *completion* time distribution of the task assigned last to the considered compute node. Once the assignment of a given task is finalized, its computed completion time distribution will be used for future assignment assessments.

### 3.3. Bootstrap approximation

This subsection presents an alternative method of evaluating $\mathbb{P}[\beta_{\min} \leq \psi_j \leq \beta_{\max}]$ known in the literature as the bootstrap method [41]. In contrast to convolution that is applicable only when $\psi_j = \sum_{i=1}^{n_j} T_{ij}$, the bootstrap procedure can be applied to *various* forms of functional dependence between local uncertainty parameters $T_{ij}$ and the local performance characteristic $\psi_j$, making it very useful in practical implementations. For example, the processing of queries by a Web server is typically done in a parallel multitasking environment, and there exists a complex functional dependence [3] between the time required to process a query and a number of currently executing threads, amount of data cached, types of requests, etc.

Suppose that for each $T_{ij}$, its execution time distribution is known and fully described with a pmf $f_{T_{ij}}(t_i)$. The pmf can be derived analytically and presented as a closed-form expression, or obtained as a result of past executions of application $i$ on compute node $j$. The latter is called a sample pmf. As the number of past executions $k$ grows, new results of executions are added, and the sample pmf, $\widehat{f}_{(k)T_{ij}}(t_i)$, constructed from these observations, converges in probability to $f_{T_{ij}}(t_i)$, i.e., $\widehat{f}_{(k)T_{ij}}(t_i) \xrightarrow{\mathbb{P}} f_{T_{ij}}(t_i)$.

$B \leftarrow$ number of bootstrap replications;
$Vboot \leftarrow$ vector of length $B$;
$Vsample \leftarrow$ vector of length $n_j$;
**for** $b = 1 : B$
  **for** $i = 1 : n_j$
    $Vsample \leftarrow$ sample $f_{T_{ij}}(t_i)$ with replacement;
  $Vboot \leftarrow g(Vsample)$;
  clear $Vsample$;
$Nsamples \leftarrow$ number of samples in $Vboot \in [\beta_{min}, \beta_{max}]$;
$\mathbb{P}[\beta_{min} \leq \psi_j \leq \beta_{max}] \approx Nsamples/B$.

Fig. 3. Pseudocode for the bootstrap procedure.

Let $\widehat{T}^*_{ij}$ denote one draw from the distribution $f_{T_{ij}}(t_i)$ (or from $\widehat{f}_{(k)T_{ij}}(t_i)$). Let $\widehat{\psi}^*_j$ be a bootstrap replication whose computation is based on a known functional dependence $g()$ between $T_{ij}$ and $\psi_j$, i.e., $\widehat{\psi}^*_j = g(\widehat{T}^*_{1j}, \ldots, \widehat{T}^*_{n_j j})$. In the bootstrap simulation step [41], $B$ bootstrap replications of $\widehat{\psi}^*_j$ can be computed, $\widehat{\psi}^*_{j,1}, \ldots, \widehat{\psi}^*_{j,B}$, and used to approximate a pmf of $\psi_j$, denoted as $\widehat{f}_{(B)\psi_j}(t)$. Thus, the probability for the local performance characteristic $\psi_j$ can be approximated as:

$$\mathbb{P}[\beta_{\min} \leq \psi_j \leq \beta_{\max}] \approx \int_{\beta_{\min}}^{\beta_{\max}} \widehat{f}_{(B)\psi_j}(t)\mathrm{d}t. \quad (6)$$

Eq. (6) assumes the existence of a monotone normalizing transformation for the $\psi_j$ distribution, and it is based on a proof of bootstrap *percentile* confidence interval [41]. An exact normalizing transformation will rarely exist, but approximate normalizing transformations may exist—which causes the probability that $\psi_j$ is in the interval $[\beta_{\min}, \beta_{\max}]$ to be not exactly equal to the integral on the right-hand side of Eq. (6). The pseudocode for the bootstrap analysis is shown in Fig. 3.

Table 1 presents the empirical data for an experiment conducted to illustrate the accuracy of the bootstrap approximation for the case where the functional dependence between $T_{ij}$ and $\psi_j$ is a summation. Table 1 captures the percent error resulted from the approximations based on Eq. (6) with respect to the exact convolution results. In the experiment, $\beta_{\min}$ was set to 0, $\beta_{\max}$ was set to the mean value of $t$ in $\widehat{f}_{(B)\psi_j(t)}(t)$—this ensures that $\beta_{\max}$ is specified in the reasonable range. All $T_{ij}$ distributions were modeled by randomly assigning a probability to each of $\Omega$ data points and normalizing the resultant pmfs. Each value in Table 1 represents the average across 100 different trials. Two trends can be identified from Table 1: (1) relative accuracy does not increase with the number of applications assigned to compute node $j$, (2) tighter approximations were obtained by increasing the number of bootstrap replications. If distributions of uncertainty parameters were closer to Gaussian distribution — which occurs often in practice — the resultant bootstrap approximations would be more precise as described in the proof of Eq. (6) [41]. There are other bootstrap approximations that may be more accurate, especially when the nature of the expected pmf of the performance metric is known. The above experiment demonstrates that the bootstrap method is capable of producing reasonable approximations. The real strength of the bootstrap is its ability to handle mutually dependent random variables. Note however that some bootstrap methods require a significant amount of computation and might be prohibitively expensive in certain distributed systems.

Table 1
Percent error resulted from bootstrap approximations

| $n_j$ | Number of bootstrap replications | | |
|---|---|---|---|
| | 100 | 1000 | 10,000 |
| 10 | 5.63 | 5.61 | 2.16 |
| 100 | 8.35 | 3.23 | 2.13 |
| 1000 | 6.52 | 2.84 | 1.04 |

## 4. Comparison with deterministic metrics

The experiments in this section seek to establish the utility of the stochastic robustness metric in distinguishing between resource allocations that perform similarly with respect to a commonly used deterministic metric, such as makespan, and the deterministic robustness metric from [2]. The simulation of the system outlined in the ARMS example of Section 1 included 1000 randomly generated resource allocations, where 128 independent applications ($N = 128$) were allocated to eight machines ($M = 8$). Each of the application execution time distributions, specific to each application–machine pair, was modeled with a pmf randomly constructed in the range [0, 40] s, inclusive. To construct each pmf, ten execution time values were uniformly spread across the range of the distribution. Each of these execution time values was assigned a probability sampled uniformly on the range (0, 1). All the application execution time distributions were subsequently normalized so that the sum of the probabilities across all the execution time values becomes equal to 1. Let $\text{mean}_{av}$ be the average value computed across the means of all constructed application execution time distributions. In the simulation, the QoS constraint $\Lambda$ was set to $\Lambda = 1.5 \times N \times \text{mean}_{av}/M$. Recall, for the ARMS example $\Lambda$ is a QoS constraint on system processing time that is used in the definition of the stochastic robustness metric given in Eq. (3). In Fig. 4, the "stochastic robustness" vertical axes correspond to the probability that the makespan will be $\leq \Lambda$. In this simulation, the deterministic robustness metric and makespan were calculated using the mean of the execution time distribution for each application–machine pair in the given allocation.

In Fig. 4(a), a comparison between the stochastic robustness metric and makespan is presented for 1000 *randomly* generated resource allocations. As can be expected, in general, resource allocations that produce a very large makespan tend to have a very small stochastic robustness metric value. However, there can be a large discrepancy between the predicted performance found using the predicted makespan, based on execution time mean values, and the predicted performance found using

the stochastic robustness metric. For example, in the figure, compare the two resource allocations labeled $A$ and $B$. If the comparison of these two resource allocations is made using the predicted makespan, allocation $A$ appears to be slightly superior to allocation $B$. However, resource allocation $B$ presents a 99.8% probability of meeting the imposed QoS constraints, whereas allocation $A$ has only a 75% probability of meeting it. In this case, using only the expected makespan to compare the two resource allocations leads to a sizable increase in risk for a modest ($\approx$5%) improvement in the expected makespan. Any of the approximately 100 resource allocations above and to the right of allocation $A$, delineated by the dashed lines in the figure, will have a higher robustness value yet higher (worse) makespan value than $A$.

In Fig. 4(b), a comparison of the stochastic robustness metric and the deterministic robustness metric is presented for 1000 randomly generated resource allocations. The deterministic robustness metric, first introduced in [2], is based on a calculation of the minimum total increase across all task execution times in the Euclidean sense that can possibly violate $\Lambda$. The results also show a number of resource allocations that have a *negative* deterministic robustness value. For the data used in this simulation study, a negative value for the deterministic robustness correlates with a low stochastic robustness value.

Compare the two resource allocations $C$ and $D$. Based on using deterministic robustness measure, allocation $D$ (with a deterministic measure of 6.13 s) is preferred over $C$ (with a deterministic measure of 3.25 s). However, under the new stochastic model, allocation $C$ (with a stochastic measure of 99.9%) is preferred over $D$ (with a stochastic measure of 75%). Thus in this case, using only the deterministic robustness metric to select a resource allocation, $D$ appears to be more robust than $C$. In contrast, the stochastic robustness metric, which accounts for the distribution of makespan outcomes, shows that allocation $C$ has a 99.9% probability of meeting the QoS constraint while allocation $D$ has only a 75% probability of meeting the QoS constraint.

Consider the sub-region identified in Fig. 4(b) with dotted lines originating from the point $D$, containing all of the points above and to the left of $D$. Each of these points in the sub-region has a higher stochastic robustness metric value than $D$ but a lower deterministic robustness metric value than $D$.

It is shown in [2] that the deterministic robustness metric, using an expected time for each task execution, provides better information for resolving a resource allocation than just a makespan. However, when execution time distributions are available, the stochastic robustness metric provides even better decision than the deterministic robustness metric. Differences between the stochastic robustness metric and the deterministic robustness metric can be explained by the fact that the stochastic robustness metric uses information about the distribution of outcomes for the resource allocation to determine robustness. In contrast, the deterministic robustness metric uses a scalar estimate of each application's execution time on each machine to determine a resource allocation's robustness. Thus, *if* the information needed for using the stochastic model is available, or can be obtained, *then* a better selection among resource allocations is possible.

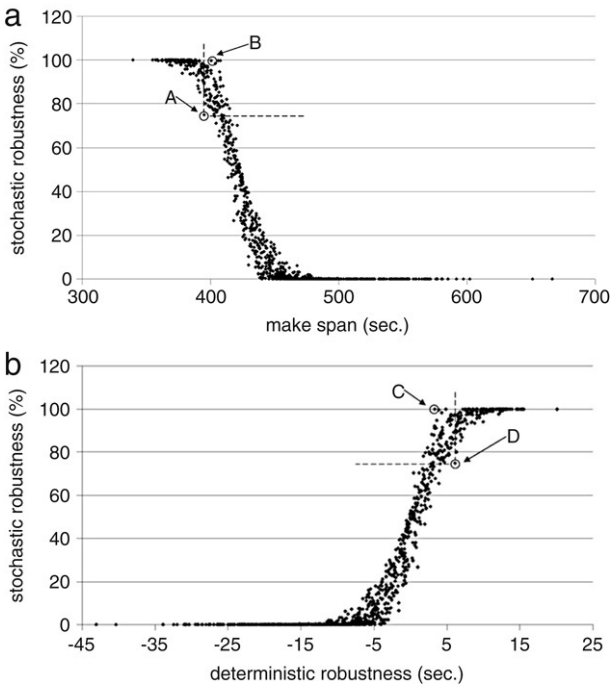


Fig. 4. A plot of stochastic robustness metric versus (a) makespan and (b) deterministic robustness for 1000 randomly generated resource allocations. The stochastic robustness metric values for allocations $A$ and $B$ exemplify the difference between the stochastic robustness metric and makespan. Similarly, the stochastic robustness metric values for allocations $C$ and $D$ exemplify the difference with the deterministic robustness metric.

## 5. Greedy heuristics

### 5.1. Overview

The stochastic robustness metric established in Section 2 can be used to solve different optimization problems. For the system illustrated in the ARMS example, the performance goal for the mapper can be formulated as follows: find resource allocations for a given set of $N$ applications on $M$ machines that allows for the minimum period $\Lambda$ between sequential data sets while maintaining a fixed user specified level of stochastic robustness $\mathbb{P}[\psi \leq \Lambda]$. Another possible formulation is to maximize the stochastic robustness in the system for a fixed user specified $\Lambda$. The research described next was focused on the first formulation.

Four greedy heuristics were designed for the problem of finding a resource allocation with respect to this objective. Greedy techniques have been adapted in many systems, e.g., [9,20,30,32], as they perform well and are capable of generating solutions relatively fast as compared to time-consuming global search heuristics, e.g., [40,42,43]. The four heuristics can be categorized based on the amount of stochastic information that each of them uses. The first two of the proposed heuristics utilize the entire spectrum of stochastic information at each stage of the decision process, as opposed to the third heuristic

$lo = t_1 \leftarrow \min\{t_{kj} \mid 1 \le k \le K_j, 1 \le j \le M\}$;
$hi = t_2 \leftarrow \max\{t_{kj} \mid 1 \le k \le K_j, 1 \le j \le M\}$;
$P \leftarrow$ specified level of $\mathbb{P}[\psi \le \Lambda]$;
**while** $\exists\ t_{kj} \in (lo, hi) \mid \{1 \le k \le K_j, 1 \le j \le M\}$
$\mathbb{P}[\psi \le \Lambda] \leftarrow \prod_{j=1}^{M} \left[ \sum_{k=1}^{K_j} p_{kj} \times \mathbf{1}(t_{kj} \in [t_1, t_2]) \right]$;
**switch** $\mathbb{P}[\psi \le \Lambda]$ **:**
$= P$ : **return**;
$> P$ : $hi \leftarrow t_2$;
$< P$ : $lo \leftarrow t_2$;
**end of switch**
$t_2 \leftarrow t_{kj} \mid \{1 \le k \le K_j, 1 \le j \le M\}$ closest to $lo + (hi - lo)/2$;
**end of while**
$\Lambda \leftarrow hi$.

Fig. 5. Pseudocode for the Period Minimization Routine (PMR).

that uses mean values in the sorting stage, and the fourth heuristic that operates using mean values only. All of the heuristics employ the Period Minimization Routine, described next, to determine the minimum $\Lambda$ supported by each resource allocation.

*Period Minimization Routine*: The PMR procedure determines the minimum possible value of $\Lambda$ for a *given* resource allocation and a *given* level of stochastic robustness. As a first step, the results of the $n_j$-fold convolutions are obtained with the FFT or bootstrap method for each compute node corresponding to the completion time (i.e., $\sum_{i=1}^{n_j} T_{ij}$) distributions expressed in a pmf form. The completion time pmf on compute node $j$ is comprised of $K_j$ impulses, where every impulse is specified by the time $t_{kj} \mid k \in [1, K_j]$, and the probability $p_{kj} \mid k \in [1, K_j]$ for $t_{kj}$ to occur.

As a second step, the minimum $\Lambda$ is determined recursively as the smallest value among $t_{kj} \mid \{1 \le k \le K_j, 1 \le j \le M\}$, such that the specified level of stochastic robustness is less than or equal to $\prod_{j=1}^{M}[\sum_{k=1}^{K_j} p_{kj} \times \mathbf{1}(t_{kj} \le \Lambda)]$, where $\mathbf{1}(condition)$ is 1 if $condition$ is true; 0 otherwise. The PMR procedure is summarized in Fig. 5.

After $Q$ steps, the PMR procedure reduces the uncertainty range by a factor $\approx (0.5)^Q$, which is the fastest possible uncertainty reduction rate. This optimality is possible because $\mathbb{P}[\psi \le \Lambda]$ is strictly increasing as the number of impulses considered for its computation grows. The notation $\Lambda(a_i, m_j)$ will be used to denote a PMR call that returns the minimum value of $\Lambda$ for the specified level of stochastic robustness when application $a_i$ is added to machine $m_j$.

### 5.2. Two-Phase heuristic

The Two-Phase heuristic is based on the principles of the Min–Min algorithm (first presented in [20], and shown to perform well in many environments, e.g., [9,29,30]). The heuristic traverses through $N$ iterations resolving an allocation of one application at each iteration. In the first phase of each iteration, the heuristic determines the best assignment (according to the performance goal) for each of the applications

**while** not all applications are mapped
**for** each unmapped application $a_i$ find the compute node $m_j$ such that
$m_j \leftarrow \text{argmin}^1\ \{\Lambda(a_i, m_j) \mid 1 \le j \le M\}$;
resolve ties arbitrarily;
from all $(a_i, m_j)$ pairs found above select pair(s) $(a_x, m_y)$ such that
$(a_x, m_y) \leftarrow \text{argmin}\ \{\Lambda(a_i, m_j) \mid$ all $(a_i, m_j)\}$;
resolve ties arbitrarily;
map $a_x$ on $m_y$;
**end of while**

Fig. 6. Pseudocode for the Two-Phase greedy heuristic. *Argmin* stands for the argument of the minimum, i.e., the value of the given argument for which the value of the given expression attains its minimum value.

**while** not all applications are mapped
**for** each unmapped application $a_i$
find the first choice compute node $m_j^{(1)}$ as
$m_j^{(1)} \leftarrow \text{argmin}\{\Lambda(a_i, m_j) \mid 1 \le j \le M\}$;
**for** all $(a_i, m_j^{(1)})$ pairs found above
**if** $m_j^{(1)} \ne m_k^{(1)},\ \forall k \ne j$, **then** map $a_i$ on $m_j^{(1)}$;
**else** find the second choice compute node $m_j^{(2)}$ as
$m_j^{(2)} \leftarrow \text{argmin}\{\Lambda(a_i, m_j) \mid (1 \le j \le M \& m_j \ne m_j^{(1)})\}$;
compute contention value $C(a_i)$ as
$C(a_i) \leftarrow \{\Lambda(a_i, m_j^{(2)}) - \Lambda(a_i, m_j^{(1)})\}$;
select unmapped application $a_x$ with maximum $C(a_i)$ as
$a_x \leftarrow \text{argmax}\{C(a_i) \mid$ all unmapped $a_i\}$;
resolve ties arbitrarily;
map $a_x$ on its first choice compute node $m_j^{(1)}$;
**end of while**

Fig. 7. Pseudocode for the Contention Resolution greedy heuristic.

left unmapped. In the second phase, it selects which application to map based on the best result found in the first phase. The Two-Phase procedure is summarized in Fig. 6.

### 5.3. Contention Resolution heuristic

The CR heuristic uses the sufferage concept introduced in [29], and used in [24]. Like the Two-Phase heuristic, in every iteration this heuristic first determines the best assignment for each of the applications left unmapped. Mapping decisions are finalized for those applications whose best choice compute nodes are unique, i.e., there are no other applications competing for these nodes. In the second phase, the most critical among the competing applications gets allocated, determined as the application with the largest difference between the two smallest $\Lambda$ values corresponding to this application's assignment to its best choice and its second best choice compute nodes, i.e., its sufferage. The Contention Resolution procedure is summarized in Fig. 7.

### 5.4. Sorting heuristic

This heuristic uses the concepts developed for the MCT algorithm that were observed to perform well in multiple

**for** each application $a_i$
  find the average $A(a_i) \leftarrow \left[\sum_{j=1}^{M} \mu(T_{ij})\right] / M$;
order applications based on their averages $A(a_i)$
according to the selected type {hi→lo, lo→hi, arbitrary};
**while** not all applications are mapped
  fetch next unmapped application $a_i$ from the ordered list;
  find the compute node $m_j$ such that $m_j \leftarrow \text{argmin}\{\Lambda(a_i, m_j) \mid 1 \leq j \leq M\}$;
  resolve ties arbitrarily;
  map $a_i$ on $m_j$;
**end of while**

Fig. 8. Pseudocode for the sorting greedy heuristic.

resource allocation schemes designed for distributed systems, e.g., [9,29]. Initially, all $N$ applications considered for mapping are sorted based on the average computed for each application across the mean values $\mu(T_{ij})$ derived from execution time distributions of $T_{ij} \mid 1 \leq j \leq M$. Three different orderings were considered in the experiments. The hi → lo ordering where applications are ranked in descending order of their averages; lo → hi ordering where applications are ranked in ascending order of their averages; and arbitrary ordering where $N$ applications are ordered randomly. Once sorting is completed, applications are fetched sequentially, each mapped on the compute node selected to provide the minimum value of the period $\Lambda$ under the imposed level of stochastic robustness. The heuristic's procedure is summarized in Fig. 8.

### 5.5. Mean load balancing heuristic

This heuristic was developed based on the concepts of the OLB algorithm discussed in [29,23]. First, the $N$ applications are sorted based on average value, as in the sorting heuristic. Then, the applications are mapped in the {hi → lo, lo → hi, arbitrary} order where the compute node with the minimum mean of its execution time distribution is selected for each allocation. The heuristic's procedure is summarized in Fig. 9.

## 6. Global search heuristics

### 6.1. Overview

Three global search heuristics were designed to find a resource allocation that optimizes the performance goal stated in Section 5.1. These heuristics are probabilistic search techniques that have been widely used in optimization research [34,43,30], artificial intelligence [19], and many other areas. The first two of the heuristics operate with a set of complete resource allocations; whereas the third heuristic iteratively changes a single complete resource allocation. As opposed to the previous greedy algorithms, where a single complete resource allocation was "constructed," iterative heuristics progress toward a final solution through modified versions of complete resource allocations. During each iteration, the existing complete resource allocation (or set of allocations) is modified and evaluated. Such an iterative search process continues until an appropriate stopping criterion is reached.

To establish a basis for the comparison of the global search heuristics and to demonstrate the performance over time for each of them, a common stopping criterion (CSC) of 150,000 calls to the PMR routine was used in this study (each PMR call corresponds to at most 1024 1-fold convolutions). It is important to note that the PMR stochastic evaluation is the most computationally intensive part of any of the algorithms as it calls for $M$ executions of $(n_j - 1)$-fold convolutions, followed by a recursive search for a minimum $\Lambda$ level.

### 6.2. Steady State genetic algorithm

The adapted genetic algorithm (GA) implementation was motivated by the Genitor evolutionary heuristic [43]. Each chromosome in the GA models a complete resource allocation as a vector of numbers of length $N$ where the $i$th element of the vector identifies the compute node assignment for application $a_i$. The order in which applications are placed in a chromosome does not play any role and can be considered arbitrary. The population size for the GA was fixed at 200 members for each iteration. The population size was chosen experimentally by varying the population size between 100 and 250 in increments of 50. For the samples tried, a value of 200 performed the best and was chosen for all trials. The initial members of the population were generated by applying the greedy Sorting heuristic presented before, in which the arbitrary ordering among applications was perturbed to have as a result different resource allocations to serve as the initial members of the population. In addition, the solution produced by the greedy Two-Phase heuristic was also added to the initial population.

The GA was implemented as a *steady state* GA, i.e., for each iteration of the GA only a single pair of chromosomes was selected for crossover. Selection for crossover was implemented as rank-based selection using a linear bias function [43] where the population of chromosomes is sorted by $\Lambda$ values. The most fit chromosome corresponds to a resource allocation with the smallest $\Lambda$ value supportable at the specified level of stochastic robustness $\theta$. Each chromosome generated by crossover or mutation is inserted into the population according

**for** each application $a_i$
  find the average $A(a_i) \leftarrow \left[\sum_{j=1}^{M} \mu(T_{ij})\right] /M$;
order applications based on their averages $A(a_i)$
according to the selected type {hi→lo, lo→hi, arbitrary};
**while** not all applications are mapped
  fetch next unmapped application $a_i$ from the ordered list;
  find the compute node $m_j$ such that $m_j \leftarrow \text{argmin}\{\mu(T_{ij}) \mid 1 \leq j \leq M\}$;
  resolve ties arbitrarily;
  map $a_i$ on $m_j$;
**end of while**

Fig. 9. Pseudocode for the Mean Load Balancing greedy heuristic.

its $\Lambda$ value such that after insertion the population remains sorted. Furthermore, the population is truncated after insertion to maintain a constant population size.

To reduce the number of duplicate chromosome evaluations, each chromosome that is trimmed from the active population is recorded in a list of known bad chromosomes referred to as the graveyard. Selecting the size of the graveyard reflected a trade-off between the time required to identify that a new chromosome was not present in the population or the graveyard and the time required to evaluate the new chromosome. Given the computer system used, the graveyard size was selected to be 20,000 chromosomes.

To maintain the selective pressure of rank-based selection, an additional constraint was placed on the population requiring each chromosome to be unique, i.e., clones are explicitly disallowed. If a chromosome produced in any iteration were to generate a clone of an individual already present in the population or the graveyard, then that clone would be discarded prior to its evaluation for insertion into the population.

The crossover operator was implemented using a two-point reduced surrogate procedure [43]. In this procedure, the selected parent chromosomes are compared to identify the chromosome entries that differ between them. Crossover points are randomly selected such that at least one element of the parent chromosomes differs between the selected crossover points as this guarantees offspring that are not clones of their parents. In addition, each generated offspring is subsequently checked for uniqueness in the population and the graveyard prior to making a call to the PMR routine that calculates the minimum $\Lambda$ value.

The final step in a single iteration of the GA is mutation. For each iteration of the GA, the mutation operator is applied to the newly generated offspring of the crossover operator. Each application assignment of the offspring is individually mutated with a probability referred to as the mutation rate. For the simulated environment, the best results were achieved using a mutation rate of 0.01. Once the application is selected, the mutation operator randomly selects a different compute node assignment from a subset of compute nodes that provide smallest means of execution times. The best results in the simulation study were achieved when the size of this subset was set to three. Following mutation a final local search procedure, conceptually analogous to the steepest descent technique, was applied to the result prior to inserting the mutated chromosome into the population.

The general idea of using a local search operator as a post-processing step to a mutation operator in a GA was used in [44] to address a flowshop problem. The following is a short synopsis of the local search procedure implemented here, conceptually similar to the coarse refinement presented as part of the GIM heuristic in [39]. The local search relies on a simple four-step procedure to minimize $\Lambda$ relative to a fixed $\theta$ level. First for a given mapping, the machine is identified with the lowest individual probability to meet $\Lambda$. On this machine, the application is identified that, if moved to a different machine, would decrease the overall $\Lambda$ the most. This requires re-evaluating the overall $\Lambda$ every time an application is tried on each machine to determine the largest improvement that can be gained by moving this application. Once an application–machine pair has been identified, the task is moved to its chosen machine. Finally, the procedure repeats from the first step until there are no application moves on the lowest probability machine that would improve $\Lambda$. For this procedure, it is assumed that $\theta < 1$; otherwise, the first step should be modified to identify the machine that finishes last, as the robustness of all machine completion times would be equal to one.

The GA procedure is summarized in Fig. 10.

### 6.3. Ant colony optimization

The Ant Colony Optimization (ACO) heuristic belongs to a class of swarm optimization algorithms, where low-level interactions between artificial (i.e., simulated) ants result in large-scale optimizations by the larger ant colony. The technique was inspired by colonies of real ants that deposit a chemical substance (pheromone) when searching for food. This substance influences the behavior of individual ants. The greater the amount of pheromone on a particular path, the larger the probability that an ant will select that path. Artificial ants in ACO behave in a similar manner by recording their chosen path in a global pheromone table.

```
generate initial population;
evaluate each chromosome;
rank population based on Λ values;
while CSC not met
   select two chromosomes from the population;
   select crossover points;
   exchange compute node assignments
   between crossover points;
   ascertain if either offspring are unique;
   for each element of each child chromosome
      generate a random number x in the range [0,1];
      if x < mutation rate
         determine 3 minimum mean execution time machines
         for the selected application;
         arbitrarily change the compute node assignment
         of the selected application;
   apply local search to each of the offspring;
   ascertain if either offspring is unique;
   insert unique offspring into population;
   trim population down to population size;
   move dead chromosomes to the graveyard;
end of while
output the best solution.
```

Fig. 10. Pseudocode for the Steady State genetic algorithm.

The ACO algorithm implemented here is a variation of the ACO algorithm design described in [14]. During ACO execution, the $N \times M$ pheromone table is maintained and updated allowing the ants to share global information about good compute nodes for each application. Let each element of the pheromone table, denoted as $\tau(a_i, j)$, represent the "goodness" of compute node $j$ for application $a_i$. At a high level, the ACO heuristic works in the following way. A certain number of ants are released to find different complete mapping solutions. Based on the mapping produced by the individual ants, the pheromone table is updated. This procedure is repeated as long as the common stopping criterion is not reached. The final mapping solution is determined by mapping each application to its highest pheromone value compute node.

The update of each pheromone table entry $\tau(a_i, j)$ involves the fitness of ant $s$, denoted as $f(s) \in (0, 1)$, determined as a relative performance of ant $s$ with respect to the performance of other ants. Let $\Lambda(s)$ be the level of $\Lambda$, obtained with a PMR call invoked at the end of the ant's mapping procedure (described below). Assuming $Q$ ants are released in an iteration, $f(s)$ is calculated as follows

$$f(s) = 1 - \frac{\Lambda(s)}{\sum_{k=1}^{Q} \Lambda(k)}. \tag{7}$$

If $\rho$ denotes a coefficient that represents pheromone evaporation, $B_s$ denotes the set of application-compute node assignments comprising the path of ant $s$, each $\tau(a_i, j)$ is updated as follows

$$\tau(a_i, j) = \rho \times \tau(a_i, j) + \sum_{s=1}^{Q} f(s) \times \mathbf{1}(a_i \text{ assigned to } j \text{ in } B_s). \tag{8}$$

As stated above, the pheromone table is updated at the end of each high-level iteration, i.e., when all ants complete their paths. Initially, all values in the pheromone table are set to 1.

At a low level, each ant heuristically "constructs" a complete mapping, and its mapping decision process balances between: (a) the performance metric and (b) the pheromone table information. The ant's mapping procedure involves two steps. In Step 1, for each unmapped application, the compute node, denoted as $j_{\text{best}}(a_i)$, is determined such that it would provide the minimum mean completion time, $\mu_{\min}(a_i)$, across $M$ compute nodes if $a_i$ was assigned to this node. The worth of application $a_i$, denoted as $\eta(a_i)$, is then determined as a result of the following normalization

$$\eta(a_i) = 1 - \frac{\mu_{\min}(a_i)}{\sum_{\text{unmapped } a_k} \mu_{\min}(a_k)}. \tag{9}$$

In Step 2, an unmapped application is stochastically selected and assigned to its $j_{\text{best}}(a_i)$ compute node. Let $\alpha$ be the scalar that controls the balance between the pheromone value and worth. The probability that ant $s$ selects application $a_i$ to be mapped next is

$$\mathbb{P}[a_i \text{ selected next}] = \frac{\alpha \times \tau(a_i, j_{\text{best}}(a_i)) + (1-\alpha) \times \eta(a_i)}{\sum_{\text{unmapped } a_k} \alpha \times \tau(a_k, j_{\text{best}}(a_k)) + (1-\alpha) \times \eta(a_k)}. \tag{10}$$

The ant's mapping procedure is repeated until all applications have been mapped.

The scalar $\alpha$ was determined experimentally by incrementing from 0 to 1 in 0.1 steps. In the simulation trials tested, the performance peak was detected with $\alpha$ equal to 0.5. The pheromone evaporation factor $\rho$ of 0.01 was determined in a similar manner. The total number of ants for each iteration was set to 50; any further increase of this number in the experiments resulted in performance degradation. Note that numerical values for all of the aforementioned parameters were empirically optimized for the simulation setup described in Section 8. The ACO procedure is summarized in Fig. 11.

### 6.4. Simulated annealing

The Simulated Annealing (SA) algorithm — also known in the literature as Monte Carlo annealing or probabilistic hill-climbing [30] — is based on an analogy taken from thermodynamics. In SA, a randomly generated solution, structured as the chromosome for GA, is iteratively modified and refined. Thus, SA in general, can be considered as an

```
initialize pheromone table;
while CSC not met
    for each ant
        while there are unmapped applications
            select application a_i according to Eq. 10;
            map application a_i to j_best(a_i) compute node;
            break ties arbitrarily;
        end of while;
        compute f(s) via PMR call;
    update pheromone table according to Eq. 8;
end of while
map each application a_i to compute node j with highest τ(a_i, j) value.
```

Fig. 11. Pseudocode for the Ant Colony Optimization heuristic.

iterative technique that operates with one possible solution (i.e., resource allocation) at a time.

To deviate from the current solution in an attempt to find a better one, SA repetitively applies the mutation operation in the same fashion as GA, including the local search. Once a new *unique* solution, denoted as $S_{\text{new}}$, is produced (SA uses the same graveyard technique as GA to determine uniqueness), a decision regarding the replacement of a previous solution with a new one has to be made. If the quality of the new solution, $\Lambda(S_{\text{new}})$, found after evaluation, is higher than the old solution, the new solution replaces the old one. Otherwise, SA uses a procedure that probabilistically allows poorer solutions to be accepted during the search process, which makes this algorithm different from other strict hill-climbing algorithms [30]. This probability is based on a system temperature, denoted $\mathbb{T}$, that decreases with each iteration. As the system temperature "cools down" it becomes more difficult for poorer solutions to be accepted. Specifically, the SA algorithm selects a sample from the range [0, 1) according to a uniform distribution. If

$$\text{random}[0, 1) > \frac{1}{1 + \exp^{(\frac{\Lambda(S_{\text{old}}) - \Lambda(S_{\text{new}})}{\mathbb{T}})}} \tag{11}$$

the new poorer resource allocation is accepted; otherwise, the old one is kept. As it follows from Eq. (11), the probability for a new solution of similar quality to be accepted is close to 50%. In contrast, the probability of poor solutions to be rejected is rather high, especially when the system temperature becomes relatively small.

After each mutation (described in the GA procedure) that successfully produces a new unique solution, the system temperature $\mathbb{T}$ is reduced to 99% of its current value. This percentage, defined as a cooling rate, was determined experimentally by varying the rate in the range of (0.9, 1] in 0.01 steps. The initial system temperature in Eq. (11) was set to $\Lambda$ of the chosen initial resource allocation.

The SA procedure is summarized in Fig. 12.

```
S_old ← initial randomly generated resource allocation;
𝕋 ← Λ(S_old);
while CSC not met
    S_new ←result of successful mutation;
    if Λ(S_new) < Λ(S_old)
        S_old ← S_new;
    else if Eq. 11 holds
        S_old ← S_new;
𝕋 ← 0.99 × 𝕋;
end of while
```

Fig. 12. Pseudocode for the Simulated Annealing heuristic.

## 7. Lower bound calculation

To evaluate the absolute performance attainable by the developed resource allocation techniques, a lower bound (LB) on the minimum period $\Lambda$ was derived based on the assumption that the specified level of the stochastic robustness metric is greater than or equal to 0.5, i.e., $\theta \geq 0.5$, which is typical for practical implementations. The process of calculating the LB involves two major steps. In the first step, a "local" lower bound on $\Lambda$ is established for a given mapping. In the second step, a unique LB is computed for all possible local lower bounds by solving a relaxed form of the Integer Linear Program formulated for the resource allocation problem.

*Step* 1: Consider a *given* complete resource allocation of $N$ applications on $M$ compute nodes. Let $\overline{\Lambda}$ denote the maximum of the means across all $M$ completion time distributions, $\mu(\sum_{i=1}^{n_j} T_{ij})$, i.e., $\overline{\Lambda} = \max\{\mu(\sum_{i=1}^{n_j} T_{ij}) \mid 1 \leq j \leq M\}$. Given that the assumed level of the stochastic robustness metric is greater than or equal to 0.5, $\overline{\Lambda}$ represents the smallest possible time period for a given mapping. To observe this, recall that

1. mean $\mu(a)$ is a "center of mass" of the distribution of random variable $a$, so that if $z$ is the compute node given

by $z = \text{argmax}\{\mu(\sum_{i=1}^{n_j} T_{ij}) \mid 1 \le j \le M\}$, then $\mathbb{P}[\psi_z \le \overline{\Lambda}] = 0.5$;

2. $\mathbb{P}[\psi_z \le \overline{\Lambda}] \ge \mathbb{P}[\psi \le \overline{\Lambda}]$ because according to Eq. (3), $\mathbb{P}[\psi \le \overline{\Lambda}]$ is computed as an $M$-product of $\mathbb{P}[\psi_j \le \overline{\Lambda}]$, where each of $M$ terms is less than or equal to one.

*Step* 2: An objective here is to determine LB, denoted as $\Lambda^*$, such that $\Lambda^* \le \min\{\overline{\Lambda} \mid \text{all possible mappings}\}$. Relying on the property that the sum of means is equal to the mean of the sums, i.e., $\sum_{i=1}^{n_j} \mu(T_{ij}) = \mu(\sum_{i=1}^{n_j} T_{ij})$, the problem of finding $\Lambda^*$ can be formulated in the following Integer Linear Programming (ILP) form.[1]

Let a *binary* decision variable $x[i, j] \mid \{1 \le i \le N; 1 \le j \le M\}$ be equal to one if application $a_i$ is assigned to compute node $j$, and equal to zero if $a_i$ is not assigned to compute node $j$. The ILP objective function can be stated as

$$\text{minimize } \Lambda^* = \max\left\{\sum_{i=1}^{N} \mu(T_{ij}) \times x[i, j] \mid 1 \le j \le M\right\}.$$

The objective function is subject to conditions (a) and (b):

$$x[i, j] \in \{0, 1\} \quad \text{for } 1 \le i \le N, 1 \le j \le M; \quad \text{(a)}$$

$$\sum_{i=1}^{N} x[i, j] = 1 \quad \text{for } 1 \le j \le M; \quad \text{(b)}$$

In addition to condition (a) explained above, condition (b) forces each application to be mapped to the system. For small-scale problems, a global optimal solution can be found for the derived ILP form in a reasonable time (e.g., by applying the Branch-and-Bound technique). However, condition (b) makes the ILP form NP-complete [31], so that for large-scale problems a Linear Programming (LP) relaxation is required to the ILP form that implies that condition (a) is relaxed to real numbers, i.e., $x[i, j] \in [0, 1] \mid \{1 \le i \le N, 1 \le j \le M\}$. Due to this relaxation, in general, an LP solution does not correspond to a valid mapping, but allows a global optimal solution to be found in *polynomial* time [18], that will be a lower bound for the ILP global optimal solution $\Lambda^*$. Note that the derived LB is tighter for stochastic robustness levels approaching 0.5; this is a result of using mean values in the LB computation.

## 8. Simulation setup

To evaluate the performance of the heuristics described above for the considered class of distributed HC systems operating on periodic data, the following approach was used to simulate a cluster-based radar system schematically illustrated in Fig. 1. The execution time distributions for twenty-eight different types of possible radar ray processing algorithms on eight ($M = 8$) heterogeneous compute nodes were generated by combining experimental data with benchmark results. The experimental data were obtained by conducting experiments on the Colorado MA1 radar [22]. These sample pmfs contain times taken to process 500 radar rays of different complexity by the "Pulse-Pair & Attenuation Correction" algorithm [7] and by the "Random Phase & Attenuation Correction" algorithm [7], both executed in non-multitasking mode on the Sun Microsystems Sun Fire V20z workstation. These execution time samples for the two algorithms were used to generate execution time pmfs for a hypothetical heterogeneous computing system. First, the execution time pmfs of 28 applications were generated for a single Sun Microsystems Sun Fire V20z workstation by scaling each of the two sample pmfs based on the relative performance results of 14 floating point benchmark applications taken from the CFP2000 suite [38]. Second, to generate data for eight heterogeneous machines, each of the 28 resultant pmfs was consequently scaled by the performance ratio of a Sun Microsystems Sun Fire V20z to each of the seven additional compute nodes.[2] The scaling was performed again based on the CFP2000 results. This method provided a means for generating a $28 \times 8$ matrix where the $ij$th element corresponds to the synthesized execution time distribution of a possible ray processing algorithm of type $i$ on compute node $j$.

A set of 128 applications ($N = 128$) was formed for each of 50 simulation trials, where for each trial the type of each application was determined by randomly sampling integers in the range $[1, 28]$. Based on 50 simulation trials, a 95% confidence interval was computed for every heuristic providing a good estimate of the average performance.

## 9. Experimental results

### 9.1. Greedy heuristics

The results of our experiments with the Greedy heuristics are presented in Fig. 13. Both the Two-Phase and Contention Resolution heuristics perform comparably and significantly outperform the Sorting and Mean Load Balancing heuristics because they utilize the entire spectrum of stochastic information at each stage of the decision process.

All of the variants of the Sorting heuristic (the results for arbitrary ordering represent the average obtained over 50 random application orderings for each trial) performed consistently better than the Mean Load Balancing heuristic variants but worse than the first two heuristics. Recall that the Sorting heuristic utilizes all of the available stochastic information to select individual task-machine pairings but relies on deterministic information to order tasks for their selection. By utilizing a task ordering process that relies on deterministic information only, the number of required convolutions to produce a mapping is drastically reduced but the quality of the mapping is also affected. For example, the Two-Phase heuristic required approximately 66,000 1-fold convolutions to produce a mapping, whereas the Sorting heuristic required only 1024 1-fold convolutions to construct a mapping. This difference in the number of convolutions directly translated into a roughly 30 times reduction in the calculations required during a simulation

[1] The ILP formulation presented below can easily be converted to a canonical ILP form [10].

[2] The seven compute nodes selected to be modeled were: Altos R510, Dell PowerEdge 7150, Dell PowerEdge 2800, Fujitsu PRIMEPOWER650, HP Workstation i2000, HP ProLiant ML370 G4, and Sun Fire X4100.

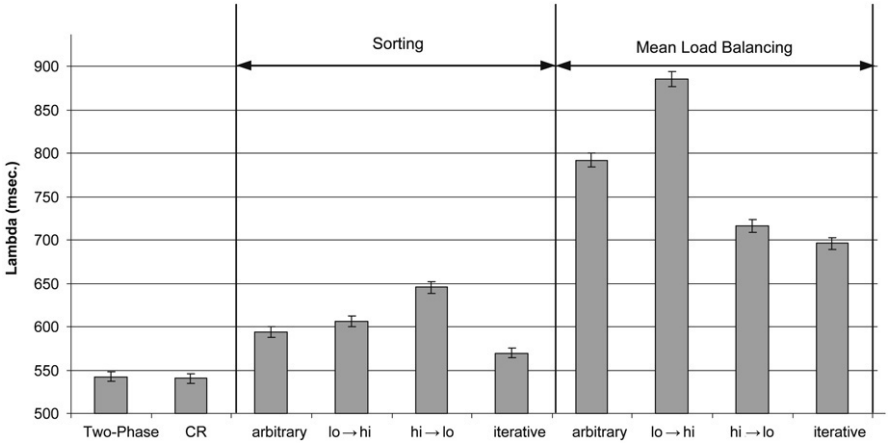


Fig. 13. A comparison of the results obtained for the described heuristics, where the minimum acceptable robustness value was set to be 0.90. The $y$-axis shows $\Lambda$ values obtained by executing the heuristics. The $\Lambda$ value for each heuristic corresponds to the average over 50 trials.

trial using the latter heuristic. Since actual timing results for our simulations is extremely system dependent, we have chosen to report the computational complexity results for our heuristics in terms of 1-fold convolutions as opposed to wall-clock time.

Finally, the Mean Load Balancing heuristic consistently performed the worst because it ignores the available stochastic information about task execution times. This results in ignoring the impact of machine heterogeneity on the completion time distributions, which is reflected in a high $\Lambda$ value. Since the Mean Load Balancing heuristic only operates with the means of execution time distributions during the mapping process, this heuristic avoided time-consuming FFT calls. This enabled this heuristic to finish in a small fraction of the time required for either Two-Phase or Contention Resolution heuristic to generate a mapping.

Once the simulation results had been collected for the heuristics, it was noticed that there was a large discrepancy in the amount of computation required to produce each of the various mappings, i.e., the first two heuristics required tens of thousands of FFT calls to produce a mapping as opposed to one-phase techniques required 1024 or less. Consequently, two new variants of the one-phase greedy algorithms that use multiple iterations, denoted in Fig. 13 as iterative, were created to increase the number of evaluated solutions to the level of Two-Phase and Contention Resolution, i.e., enable these variants to utilize roughly the same amount of computation to produce a mapping.

In both iterative greedy variants, a random restart step was introduced so that after a mapping is produced a new random ordering is generated and the heuristic is executed again. Upon completion of each iteration the resultant mapping is compared against the best mapping found so far by previous iterations. If the new mapping is an improvement on the best mapping, then it is retained as the new best mapping, otherwise it is discarded.

The results of the iterative variants are plotted in Fig. 13. As can be expected, the results of both iterative greedy approaches demonstrated some improvement over their non-iterative versions. However, the iterative version of the Sorting greedy heuristic performed worse than the Two-Phase heuristic (the confidence intervals of the two do not overlap) but is a marked improvement over the corresponding non-iterative greedy version. The average $\Lambda$ over 50 trials of the Two-Phase heuristic was 542.5 ms; whereas the average $\Lambda$ over 50 trials of the iterative version of the Sorting greedy heuristic was 569.7 ms—each had a confidence interval of 7 ms. The performance demonstrated by the iterative version of the Mean Load Balancing heuristic was still significantly worse than the performance of the other heuristics.

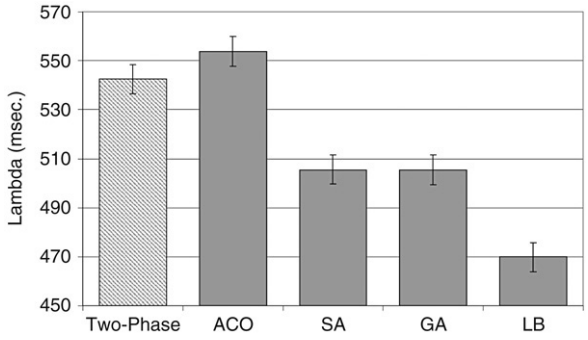


Fig. 14. A comparison of the results obtained for the global search heuristics, where the minimum acceptable robustness value was set to 0.90. The $y$-axis corresponds to a $\Lambda$ value obtained by executing the corresponding heuristics. The $\Lambda$ value for each heuristic corresponds to the average over 50 trials, while the error bars correspond to 95% confidence intervals.

### 9.2. Global search heuristics

The results of the global search heuristic simulations are presented in Fig. 14. Both the GA and SA heuristics were able to improve upon the results of the Two-Phase heuristic by more than 7% with respect to the absolute performance and by 50% with respect to the derived LB. However, the ACO procedure was unable to improve upon the results of the Two-Phase heuristic but was able to produce results such that the confidence intervals of the ACO and Two-Phase results are overlapping.

Both the GA and SA heuristics performed comparably in this simulation environment. The success of the SA procedure and the near overlap of the SA and GA results may suggest that the local search procedure used in the mutation operator by both GA and SA is responsible for their marked improvement over Two-Phase. Additional experiments were conducted using the GA without utilizing local search, and although the simple GA was able to improve the average result of the Two-Phase heuristic by almost 2%, the improvement was not statistically significant.

The ACO heuristic was unable to improve upon the results of the Two-Phase heuristic. The effectiveness of ACO in this environment relies on the repetitive application of a constructive heuristic within a high-level iteration to update the pheromone table. This might suggest that using only the mean values of the execution time distributions to construct ant solutions is insufficient. Instead of operating with mean values, intermediate minimum levels of $\Lambda$ could be computed through PMR calls. However, this would dramatically increase the number of evaluations required by ACO to produce ants in each iteration. In so doing, the number of high-level iterations that the ACO would be able to complete within the common stopping criterion would be significantly reduced.

The success of combining a simple local search with GA and SA suggest that a more exhaustive local search may be worth investigating in future work. The more exhaustive local search might consider swapping applications between compute nodes in addition to moving applications between compute nodes. Although the introduction of swapping will increase the number of evaluations required to complete the local search procedure, it may lead to an improved result over the current approach to local search.

## 10. Related literature

A universal framework for defining robust resource allocations in heterogeneous computing systems was addressed in [2]—prior work performed by our research team. This work referred to a resource allocation's tolerance to uncertainty as the robustness of that resource allocation. In [2], a four-step procedure is established for deriving a deterministic robustness metric. The first step in defining a robustness metric requires quantitatively describing what makes the system robust. This description establishes the required QoS level that must be delivered to refer to the system as robust—essentially bounding the acceptable variation in system performance. A pair of values, $\beta_{\min}$ and $\beta_{\max}$ that bound each performance feature must be identified, quantitatively defining the tolerable variation in each of the performance features.

In the second step, all modeled system and environmental parameters that may impact the system's ability to deliver acceptable QoS are identified. These parameters are referred to as the perturbation parameters of the system. In our new stochastic approach, each perturbation parameter, or uncertainty parameter, is modeled as a random variable fully described by a pmf. In this way, all possible values of the considered perturbation parameters, and their associated probabilities, are included in the calculation of the stochastic robustness metric. Our new approach differs from that in [2], where a single deterministic estimated value for each of the identified perturbation parameters is used.

In the third step, the impact of the identified perturbation parameters on the system's performance features is defined. This requires identifying a function that maps a given vector of perturbation parameters to a value for the performance feature of the system. Similarly in our new stochastic environment, this involves defining the functional dependence between the input random variables and the given performance feature. However, in our new model this involves more complex computations to combine random variables.

Finally, in the fourth step, the previously identified relation is evaluated to quantify the robustness. As a measure of robustness, the authors in [2] use the "minimum robustness radius" that relies on a deterministic performance characteristic. Furthermore, it assumes there is no *a priori* information available about the relative likelihood or magnitude of change for each perturbation parameter. Thus, the minimum robustness radius is used in a deterministic worst-case analysis. In our new stochastic model, more information regarding the variation in the perturbation parameters is assumed known. Representing the uncertainty parameters of the system as stochastic variables enables the robustness metric in the stochastic model to account for all possible outcomes for the performance of the system. This added knowledge comes at a computational cost. The stochastic robustness metric requires more information and is far more complex to calculate than its deterministic counterpart. To handle the computational complexity, we considered the FFT and bootstrap approximation methods that greatly simplify the required calculations.

In [8], the problem of robust resource allocation was addressed for Directed Acyclic Graphs (DAGs). Robustness in this work was defined in terms of a schedule's ability to tolerate an increase in execution time in components of a DAG, i.e., applications and data transfers, without increasing the total execution time of the DAG. Quantitatively, robustness was measured by a "critical", i.e., the smallest, slack among all components that comprise a given DAG. Once the metric was established, the authors provided design methods for generating resource allocations with maximized critical slack. Our stochastic robustness metric framework is more generally applicable, allowing for any definition of QoS and able to incorporate any identified uncertainty parameters.

Our methodology relies heavily on an ability to model the uncertainty parameters as stochastic variables. Several previous efforts have established a variety of techniques for determining the stochastic behavior of application execution times [6,12,28]. In [6], the authors also present a means for combining stochastic task representations to determine task completion time distributions. Our work leverages this method of combining independent task execution time distributions and extends it by defining a means for measuring the robustness of a resource allocation against an expressed set of QoS constraints.

In [21], a procedure for predicting task execution times is presented. The authors introduce a methodology for defining data driven estimates in a heterogeneous computing environment based on nonparametric inference. The proposed method is applied to the problem of generating an application execution time prediction given a set of observations of that application's past execution times on different compute nodes. The model defines an application execution time random variable as the combination of two elements. The first element corresponds to a vector of known factors that have an impact on the execution time of the application and is considered to be a mean of the execution time random variable. A second element accounts for all unmodeled factors that may impact the execution time of an application and is used to compute a sample variance. Potentially, this method can be extended to determine probability density functions describing the input random variables in our framework.

The deterministic robustness metric established for distributed systems in [2] was used in multiple heuristics approaches presented in [39]. Two variations of robust mapping of independent tasks to machines were studied in that research. In the fixed machine suite variation, six static heuristics were presented that maximize the robustness of a mapping against aggregate errors in the execution time estimates. A variety of evolutionary algorithms, e.g., Genitor and Memetic Algorithm, demonstrate higher performance as compared to the greedy heuristics. However, greedy heuristics required significantly less time to complete a mapping. A similar trade-off was observed for another variation where a set of machines must be selected under a given dollar cost constraint that will maximize the robustness of a mapping. In our study, greedy heuristics applied in a stochastic domain did not demonstrate four orders of magnitude speedup over evolutionary search algorithms due to a substantial number of calls for a convolution routine required at each step of a mapping "construction" process.

In [13], the authors present a derivation of the makespan problem that relies on a stochastic representation of task execution times. This work is the only other effort that we know of that explicitly uses a stochastic approach to scheduling task execution in a distributed computing system. The authors demonstrate that their stochastic approach to scheduling can significantly reduce the actual simulated system makespan as compared to some well known scheduling heuristics that are founded in a deterministic approach to modeling task execution times. The heuristics presented in that study were adapted to the stochastic domain and used to minimize the expected system makespan given a stochastic model of task execution times, i.e., the fitness metric there was based on the first moment of random variables. The success of the authors' Genetic Algorithm applied to this problem domain was another motivating factor for our selection of a Genetic Algorithm in this study. As shown in [13], this approach works well for unconstrained optimization problems; however, in our study, the imposed QoS constraint (e.g., $\Lambda$ in the ARMS example of Section 1) in the distributed system makes the optimization problem constrained calling for other methods. Therefore, our emphasis is on quantitatively comparing one resource allocation to another by deriving a metric for the resource allocation's robustness, i.e., the probability to deliver on expressed QoS constraints.

## 11. Conclusion

This paper proposes a stochastic framework that allows for evaluation and generation of robust resource allocations in distributed heterogeneous computer systems operating in uncertain environments. As a basis for this framework, a new stochastic robustness metric was established mathematically. Given the raw volume of computation required to compute this metric, the bootstrap approximation and FFT computational methods were explored to aid the practitioner to apply this approach in different real-world scenarios. An example of the utility of the new metric was evaluated in the simulated environment based on distinguishing among resource allocations that perform similarly with respect to a commonly used deterministic metric, such as makespan, and the deterministic robustness metric presented in [2].

In the second part of this work, the new stochastic robustness metric was integrated into a set of greedy and global search heuristics designed for heterogeneous clusters operating on periodic data sets. The goal was to generate a resource allocation that allows for the minimum time period between sequential sensor outputs in a simulated radar system, and to guarantee a specified level of probability that data processing is completed in time.

The Two-Phase and CR greedy heuristics developed in this study utilized the entire spectrum of the available stochastic information. These heuristics significantly outperformed Sorting and Mean Load Balancing heuristics, as the stochastic information in the last two was replaced with mean values completely or in the first phase. Furthermore, greedy heuristics were rather time-consuming when applied in the stochastic domain due to multiple calculations of the resultant probability mass functions. Thus, it was reasonable to compare the performance of the greedy heuristics against global search algorithms. Three global search algorithms adapted in this work, i.e., GA, SA, and ACO, were tested under the same stopping criterion. Multiple parameters pertaining to each algorithm were set up for the highest efficiency in the given environment. A comparison analysis against the best greedy results and the lower bound, obtained by solving the relaxed ILP form, revealed the great potential for the GA and SA algorithms to manage efficiently resource allocations in distributed heterogeneous systems operating under uncertainty.

## Acknowledgments

This research was supported by NSF under grant No. CNS-0615170, by the DARPA Information Exploitation Office under contract No. NBCHC030137, by the IBM Ph.D. Fellowship Program, by the Colorado State University Center for Robustness in Computer Systems (funded by the Colorado Commission on Higher Education Technology Advancement Group through the Colorado Institute of Technology), and by the Colorado State University George T. Abell Endowment.

Table 2
Glossary of notation

| | |
|---|---|
| $S^k$ | $k$th string specified by a sequence of $n_k$ applications $\{a_1^k a_2^k ... a_{n_k}^k\}$ |
| $W[k]$ | Worth factor of $k$th string |
| $P[k]$ | Period of time between sequential raw data sets processed by $k$th string |
| $m[i,k]$ | Machine to which application $a_i^k$ is assigned |
| $t_{comp}^k[i]$ | Estimated computation time for application $a_i^k$ on machine $m[i,k]$ |
| $t_{tran}^k[i]$ | Estimated time to transfer output $O^k[i]$ from $a_i^k$ to $a_{i+1}^k$ in string $S^k$ |
| $\bar{t}^k[i,j]$ | Nominal data set processing time of $a_i^k$ executing on machine $j$ |
| $\bar{u}^k[i,j]$ | Average CPU utilization of machine $j$ when $a_i^k$ processes a nominal data set |
| $U^{machine}[j]$ | Utilization of machine $j$ |
| $b[j_1,j_2]$ | Time to transmit one bit of data from machine $j_1$ to machine $j_2$ |
| $U^{route}[j_1,j_2]$ | Utilization of the communication route from machine $j_1$ to machine $j_2$ |
| $M$ | Number of heterogeneous machines in the system |
| $\Lambda$ | System slackness, i.e., the minimum utilization capacity remaining across all computation and communication resources |
| $\bar{t}_{av}^k[i]$ | Average nominal execution time of $a_i^k$ computed across $M$ machines |
| $\bar{u}_{av}^k[i]$ | Average nominal CPU utilization of $a_i^k$ computed across $M$ machines |
| $Q$ | Total number of strings considered for mapping |

Table 3
Acronyms

| | |
|---|---|
| ARMS | Adaptive and Reflective Middleware Systems |
| IMR | Incremental Mapping Routine |
| PSG | Permutation Space Genitor-based heuristic |
| ILP | Integer Linear Programming |
| LP | Linear Programming (can be achieved by relaxing an integer restriction in the corresponding ILP form) |
| UB | Upper Bound |
| B&B | Branch-and-Bound algorithm |

Preliminary portions of this material were presented at the Eighth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2006) [35], the 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2006) [37], and the 2006 International Conference on Parallel Processing (ICPP 2006) [36]. The authors thank David Janovy for his valuable comments.

## Appendix

See Tables 2 and 3.

## References

[1] S. Ali, T.D. Braun, H.J. Siegel, A.A. Maciejewski, N. Beck, L. Bölöni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, Parallel, Distributed, and Pervasive Computing, ser. Advances in Computers, Elsevier, Amsterdam, The Netherlands, 2005, pp. 91–128.

[2] S. Ali, A.A. Maciejewski, H.J. Siegel, J.-K. Kim, Measuring the robustness of a resource allocation, IEEE Transactions on Parallel and Distributed Systems 15 (7) (2004) 630–641.

[3] M.F. Arlitt, C.L. Williamson, Internet Web servers: Workload characterization and performance implications, IEEE/ACM Transactions on Networking 5 (5) (1997) 631–645.

[4] Adaptive and Reflective Middleware Systems (ARMS), Accessed Dec. 10, 2005. [Online]. Available: http://dtsn.darpa.mil/ixo/ixo_FeatureDetail.asp?id=6#.

[5] L.A. Barroso, J. Dean, U. Holzle, Web search for a planet: The Google cluster architecture, Micro IEEE 23 (2) (2003) 22–28.

[6] G. Bernat, A. Colin, S.M. Peters, WCET analysis of probabilistic hard real-time systems, in: Proceedings of 23rd IEEE Real-Time Systems Symposium, RTSS '02, 2002.

[7] N. Bharadwaj, V. Chandrasekar, Waveform design for CASA X-band radars, in: Proceedings of 32nd Conference on Radar Meteorology of American Meteorology Society, Oct. 2005.

[8] L. Bölöni, D. Marinescu, Robust scheduling of metaprograms, Journal of Scheduling 5 (5) (2002) 395–412.

[9] T.D. Braun, H.J. Siegel, N. Beck, L. Bölöni, R.F. Freund, D. Hensgen, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing 61 (6) (2001) 810–837.

[10] E.K.P. Chong, S.H. Zak, An Introduction to Optimization, 2nd ed., John Wiley, New York, NY, 2001.

[11] E.G. Coffman (Ed.), Computer and Job-Shop Scheduling Theory, John Wiley & Sons, New York, NY, 1976.

[12] L. David, I. Puaut, Static determination of probabilistic execution times, in: Proceedings of 16th Euromicro Conference on Real-Time Systems, ECRTS '04, Jun. 2004.

[13] A. Dogan, F. Ozguner, Genetic algorithm based scheduling of meta-tasks with stochastic execution times in heterogeneous computing systems, Cluster Computing 7 (2) (2004) 177–190.

[14] M. Dorigo, L.M. Gambardella, Ant colony system: A cooperative learning approach to the traveling salesman problem, IEEE Transactions on Evolutionary Computation 1 (1) (1997) 53–66.

[15] M.M. Eshaghian (Ed.), Heterogeneous Computing, Artech House, Norwood, MA, 1996.

[16] D. Fernandez-Baca, Allocating modules to processors in a distributed system, IEEE Transactions on Software Engineering 15 (11) (1989) 1427–1436.

[17] B.A. Forouzan, Data Communications and Networking, 4th ed., McGraw-Hill Science, New York, NY, 2006.

[18] C.C. Gonzaga, Path-following methods for linear programming, SIAM Review 34 (2) (1992) 167–224.

[19] J.H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, MIT Press, Cambridge, MA, 1992.

[20] O.H. Ibarra, C.E. Kim, Heuristic algorithms for scheduling independent tasks on non-identical processors, Journal of the ACM 24 (2) (1977) 280–289.

[21] M.A. Iverson, F. Ozguner, L. Potter, Statistical prediction of task execution times through analytical benchmarking for scheduling in a heterogeneous environment, IEEE Transactions on Computers 48 (12) (1999) 1374–1379.

[22] F. Junyent, V. Chandrasekar, D. McLaughlin, S. Frasier, E. Insanic, R. Ahmed, N. Bharadwaj, E. Knapp, L. Krnan, R. Tessier, Salient features of radar nodes of the first generation NetRad system, in: Proceedings of IEEE International Geoscience and Remote Sensing Symposium 2005, IGARSS '05, Jul. 2005, pp. 420–423.

[23] J.-K. Kim, H.J. Siegel, A.A. Maciejewski, R. Eigenmann, Dynamic mapping in energy constrained heterogeneous computing systems, in: Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS'05, Apr. 2005.

[24] J.-K. Kim, S. Shivle, H.J. Siegel, A.A. Maciejewski, T. Braun, M. Schneider, S. Tideman, R. Chitta, R.B. Dilmaghani, R. Joshi, A. Kaul, A. Sharma, S. Sripada, P. Vangari, S.S. Yellampalli, Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment, Journal of Parallel and Distributed Computing 67 (2) (2007) 154–169.
[25] A. Kumar, R. Shorey, Performance analysis and scheduling of stochastic fork-join jobs in a multicomputer system, IEEE Transactions on Parallel and Distributed Systems 4 (10) (1993).
[26] C. Leangsuksun, J. Potter, S. Scott, Dynamic task mapping algorithms for a distributed heterogeneous computing environment, in: Proceedings of 4th IEEE Heterogeneous Computing Workshop, HCW '95, Apr. 1995, pp. 30–34.
[27] A. Leon-Garcia, Probability & Random Processes for Electrical Engineering, Addison Wesley, Reading, MA, 1989.
[28] Y.A. Li, J.K. Antonio, H.J. Siegel, M. Tan, D.W. Watson, Determining the execution time distribution for a data parallel program in a heterogeneous computing environment, Journal of Parallel and Distributed Computing 44 (1) (1997) 35–52.
[29] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, R.F. Freund, Dynamic mapping of a class of independent tasks onto heterogeneous computing systems, Journal of Parallel and Distributed Computing 59 (2) (1999) 107–121.
[30] Z. Michalewicz, D.B. Fogel, How to Solve It: Modern Heuristics, Springer-Verlag, New York, NY, 2000.
[31] K.G. Murty, S.N. Kabadi, Some NP-complete problems in quadratic and nonlinear programming, Mathematical Programming 39 (2) (1987) 117–129.
[32] G. Nemhauser, L.A. Wolsey, Integer and Combinatorial Optimization, John Wiley & Sons, New York, NY, 1999.
[33] C.L. Phillips, J.M. Parr, E.A. Riskin, Signals, Systems, and Transforms, Pearson Education, Upper Saddle River, NJ, 2003.
[34] V. Shestak, E.K.P. Chong, A.A. Maciejewski, H.J. Siegel, L. Benmohamed, I.-J Wang, R. Daley, Resource allocation for periodic applications in a shipboard environment, in: Proceedings of 14th Heterogeneous Computing Workshop, HCW 2005. In the Proceedings of 19th International Parallel and Distributed Processing Symposium, IPDPS 2005, Apr. 2005, pp. 122–127.
[35] V. Shestak, J. Smith, A.A. Maciejewski, H.J. Siegel, Iterative algorithms for stochastically robust static resource allocation in periodic sensor driven clusters, in: Proceedings of 8th IASTED International Conference on Parallel and Distributed Computing and Systems, PDCS 2006, Nov. 2006, pp. 166–174.
[36] V. Shestak, J. Smith, A.A. Maciejewski, H.J. Siegel, A stochastic approach to measuring the robustness of resource allocations in distributed systems, in: Proceedings of International Conference on Parallel Processing, ICPP–06, Aug. 2006, pp. 459–470.
[37] V. Shestak, J. Smith, R. Umland, J. Hale, P. Moranville, A.A. Maciejewski, H.J. Siegel, Greedy approaches to static stochastic robust resource allocation for periodic sensor driven distributed systems, in: Proceedings of the 2006 International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'06, Jun. 2006.
[38] Standard performance evaluation corporation. Accessed Feb. 6, 2006. [Online]. Available: http://www.spec.org/.
[39] P. Sugavanam, H.J. Siegel, A.A. Maciejewski, M. Oltikar, A. Mehta, R. Pichel, A. Horiuchi, V. Shestak, M. Al-Otaibi, Y. Krishnamurthy, S. Ali, J. Zhang, M. Aydin, P. Lee, K. Guru, M. Raskey, A. Pippin, Robust static allocation of resources for independent tasks under makespan and dollar cost constraints, Journal of Parallel and Distributed Computing 67 (4) (2007) 400–416.
[40] L. Wang, H.J. Siegel, V.P. Roychowdhury, A.A. Maciejewski, Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach, Journal of Parallel and Distributed Computing 47 (1) (1997) 8–22.
[41] L. Wasserman, All of Statistics: A Concise Course in Statistical Inference, Springer Science+Business Media, New York, NY, 2005.
[42] J.P. Watson, L. Barbulescu, L.D. Whitley, Contrasting structured and random permutation flow-shop scheduling problems: Search-space topology and algorithm performance, INFORMS Journal on Computing 14 (2) (2002) 98–123.
[43] D. Whitley, The genitor algorithm and selective pressure: Why rank-based allocation of reproductive trials is best, in: Proceedings of 3rd International Conference on Genetic Algorithms, Jun. 1989, pp. 116–121.
[44] T. Yamada, C. Reeves, Permutation flowshop scheduling by genetic local search, in: Genetic Algorithms in Engineering Systems: Innovations and Applications, 1997, pp. 232–238.

**Vladimir Shestak** is pursuing a Ph.D. degree from the Department of Electrical and Computer Engineering at Colorado State University, where he has been a Research Assistant since August 2003. He is currently a Software Engineer at InfoPrint Solution Company, Boulder, CO. He received his M.S. degree in Computer Engineering from New Jersey Institute of Technology in May 2003. His research interests include resource management within distributed computing systems, algorithm parallelization, and computer network design and optimization. Home page www.engr.colostate.edu/~chestak.

**Jay Smith** is currently a Senior Software Engineer in the Software and Services Group within Digitalglobe. In addition to his duties within Digitalglobe, Jay is currently pursuing his Ph.D. in Electrical and Computer Engineering at Colorado State University under Prof. H.J. Siegel. His research interests include resource management within distributed computing systems, algorithm parallelization, and optimization. He is a member of the IEEE and the ACM.

**Anthony A. Maciejewski** received BSEE, MS, and Ph.D. degrees from Ohio State University in 1982, 1984, and 1987, respectively. From 1988 to 2001, he was a professor of Electrical and Computer Engineering at Purdue University, West Lafayette. He is currently a Professor and Department Head of Electrical and Computer Engineering at Colorado State University. He is a Fellow of the IEEE. His research interests include robotics and high performance computing. A complete vita is available at: http://www.engr.colostate.edu/~aam.

**Howard Jay Siegel** was appointed the Abell Endowed Chair Distinguished Professor of Electrical and Computer Engineering at Colorado State University in 2001, where he is also a Professor of Computer Science and Director of the university-wide Information Science and Technology Center (ISTeC). From 1976 to 2001, he was a professor at Purdue University. He is a Fellow of the IEEE and a Fellow of the ACM. He received two B.S. degrees (1972) from the Massachusetts Institute of Technology (MIT), and his M.A. (1974), M.S.E. (1974), and Ph.D. (1977) degrees from Princeton University. He has co-authored over 350 technical papers. His research interests include heterogeneous parallel and distributed computing, parallel algorithms, and parallel machine interconnection networks. Home page www.engr.colostate.edu/~hj.