J. Parallel Distrib. Comput. 68 (2008) 1070–1080

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Static heuristics for robust resource allocation of continuously executing applications☆

Shoukat Ali [a], Jong-Kook Kim [b,*], Howard Jay Siegel [c,d], Anthony A. Maciejewski [c]

[a] Intel Corporation, Platform Validation and Enabling, Folsom, CA 95630, USA

[b] Department of Electronics and Computing Engineering, College of Engineering, Korea University, 5-ka Anam-dong, Sungbuk-gu, Seoul, 136-701, South Korea

[c] Colorado State University, Department of Electrical and Computer Engineering, Fort Collins, CO 80523-1373, USA

[d] Colorado State University, Department of Computer Science, Fort Collins, CO 80523-1373, USA

ARTICLE INFO

Article history:
Received 30 August 2006
Received in revised form
25 May 2007
Accepted 9 December 2007
Available online 16 March 2008

Keywords:
Heterogeneous distributed computing
Genetic algorithm
Resource allocation
Robustness
Task scheduling
Shipboard computing
Simulated annealing
Static mapping

ABSTRACT

We investigate two distinct issues related to resource allocation heuristics: robustness and failure rate. The target system consists of a number of sensors feeding a set of heterogeneous applications continuously executing on a set of heterogeneous machines connected together by high-speed heterogeneous links. There are two quality of service (QoS) constraints that must be satisfied: the maximum end-to-end latency and minimum throughput. A failure occurs if no allocation is found that allows the system to meet its QoS constraints. The system is expected to operate in an uncertain environment where the workload, i.e., the load presented by the set of sensors, is likely to change unpredictably, possibly resulting in a QoS violation. The focus of this paper is the design of a static heuristic that: (a) determines a *robust* resource allocation, i.e., a resource allocation that maximizes the allowable increase in workload until a run-time reallocation of resources is required to avoid a QoS violation, and (b) has a very low *failure rate* (i.e., the percentage of instances a heuristic fails). Two such heuristics proposed in this study are a genetic algorithm and a simulated annealing heuristic. Both were "seeded" by the best solution found by using a set of fast greedy heuristics.

1. Introduction

Parallel and distributed systems may operate in an environment where certain system performance features degrade due to unpredictable circumstances, such as sudden machine failures, higher than expected system load, or inaccuracies in the estimation of system parameters. An important question then arises: given a system design, what extent of departure from the assumed circumstances will cause a performance feature to be unacceptably degraded? That is, *how robust* is the system?

Why should we care about answering the above question? A literature search for "robust networks" or "robust systems" or "robust computing" gives one reason. System builders are becoming increasingly interested 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[1] (2002) concluded that the "Issues are... being able to quantify and design for robustness..". There are many other projects of similar nature at other schools and organizations (including IBM and Raytheon, which have been working on robustness in industry computing environments).

For a particular class of heterogeneous computing (*HC*) systems, we investigate two distinct issues related to resource allocation heuristics: robustness and failure rate. The target HC system consists of a number of sensors feeding a set of heterogeneous applications continuously executing on a set of heterogeneous

☆ This research was supported by the National Science Foundation under Contract No. CNS-0615170, by the DARPA/ITO Quorum Program through the Office of Naval Research under Grant No. N00014-00-1-0599, 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. Some of the equipment used was donated by Intel and Microsoft.

* Corresponding author.
E-mail addresses: shoukat.ali@intel.com (S. Ali), jongkook.kim@gmail.com, jongkook@korea.ac.kr (J.-K. Kim), hj@colostate.edu (H.J. Siegel), aam@colostate.edu (A.A. Maciejewski).

[1] http://www.ipam.ucla.edu/programs/cnrob/.

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.12.007

Table 1
Glossary of notation

$\mathcal{M}$	the set of machines
m_j	j-th machine in the system
$\mathcal{A}$	the set of applications
a_i	i-th application in the system
$\mathcal{P}$	the set of paths
$\mathcal{P}_k$	k-th path
$R(a_i)$	for application a_i in $\mathcal{P}_k$, the output data rate of the driving sensor for $\mathcal{P}_k$
$\alpha_{k,i}$	i-th application in the k-th path
$L_k^{\max}$	maximum end-to-end latency constraint for $\mathcal{P}_k$
$\boldsymbol{\sigma}$	the set of sensors = $\left[\sigma_1 \cdots \sigma_{\lvert\boldsymbol{\sigma}\rvert}\right]$
$\boldsymbol{\lambda}$	system workload = $\left[\lambda_1 \cdots \lambda_{\lvert\boldsymbol{\sigma}\rvert}\right]^{\mathrm{T}}$
$\boldsymbol{\lambda}^{\text{init}}$	the initial value of $\boldsymbol{\lambda}$
$C_{ij}(\boldsymbol{\lambda})$	estimated time to compute for a_i on m_j for a given $\boldsymbol{\lambda}$
$T_{ij}^c(\boldsymbol{\lambda})$	the computation time for a_i on machine m_j when a_i shares this machine with other applications
$M_{ij}(\boldsymbol{\lambda})$	the size of the message data sent from a_i to a_j for a given $\boldsymbol{\lambda}$
$T_{ij}^t(\boldsymbol{\lambda})$	the time to transfer the output data from a_i to a_j
$\Delta\boldsymbol{\Lambda}$	robustness of a mapping
$\Delta\boldsymbol{\Lambda}_{ij}^{\mathrm{T}}$	robustness of assignment of a_i to m_j with respect to throughput constraint
$\Delta\boldsymbol{\Lambda}_k^L$	robustness of assignment of applications in $\mathcal{P}_k$ with respect to $L_k^{\max}$
$\Delta\boldsymbol{\Lambda}^{\mathrm{T}^*}(a_i, m_j)$	the value of $\Delta\boldsymbol{\Lambda}_{ij}^{\mathrm{T}}$ *if* application a_i is mapped on m_j
$\Delta\boldsymbol{\Lambda}^*(a_i, m_j)$	the value of $\Delta\boldsymbol{\Lambda}$ *if* application a_i is mapped on m_j

machines connected together by high-speed heterogeneous links. There are two quality of service (*QoS*) constraints that must be satisfied: the maximum end-to-end latency and minimum throughput. A failure occurs if no allocation is found that allows the system to meet its QoS constraints. The system is configured with an initial *mapping* (i.e., allocation of resources to applications) that is used when the system is first started. The system is expected to operate in an uncertain environment where the workload, i.e., the load presented by the set of sensors, is likely to change unpredictably, possibly resulting in a QoS violation. The focus of this paper is the design of a static heuristic that: (a) determines a *robust* resource allocation (i.e., a resource allocation that maximizes the allowable increase in workload until a run-time reallocation of resources is required to avoid a QoS violation), and (b) has a very low *failure rate* (i.e., the percentage of instances a heuristic fails). Two such heuristics proposed in this study are a genetic algorithm and a simulated annealing heuristic. Both were "seeded" by the best solution found by using a set of fast greedy heuristics.

One example of the kind of HC system being considered here is the Navy's planned future approach to shipboard computing called the "total shipboard computing environment". Much research is being done by the Navy and DARPA to design and develop this approach. A working prototype called the "High Performance Distributed Computing Program (HiPer-D) environment" exists at the Naval Surface Warfare Center, Dahlgren Division [11,23]. Our research was done in conjunction with this group.

There are other application domains where this model applies. Examples include monitoring the vital signs and medicine administration for a patient, recording scientific experiments, and surveillance for homeland security.

For all of systems, robustness of the initial mapping is an important concern. Generally, these systems operate in an environment that undergoes unexpected changes, e.g., in the system workload, which may cause a QoS violation. Therefore, even though a good initial mapping of applications may ensure that no QoS constraints are violated when the system is first put in operation, run-time mapping approaches may be needed to reallocate resources at run time to avoid QoS violations (e.g., [23]).

The general goal of this paper is to delay the *first re-mapping* of resources required at run time to prevent QoS violations due to variations in the amount of the workload generated by the changing sensor outputs. This work uses a generalized performance metric that is suitable for evaluating an initial mapping for such robustness against increases in the workload (a formal definition of robustness and a general procedure to derive it are given in [3]). The *initial mapping problem* is defined as finding a static mapping (i.e., one found in an off-line planning phase) of a set of applications onto a suite of machines to maximize the *robustness* against workload, defined as the maximum allowable increase in system workload until run-time re-mapping of the applications is required to avoid a QoS violation. Our contributions in this paper include designing and developing mapping heuristics so as to optimize the robustness, and evaluating the relative performance of these heuristics for the intended dynamic distributed HC system. The mapping problem has been shown, in general, to be NP-complete [12,16]. Thus, the development of heuristic techniques to find near-optimal mappings is an active area of research (many examples are given in [1]).

Section 2 develops models for the applications and the hardware platform. These models are then used in Section 3 to derive expressions for computation and communication times. Section 4 presents a quantitative measure of the robustness of a given mapping of applications to machines. Heuristics to solve the initial mapping problem are described in Section 5. The simulation experiments and the evaluation of the heuristics is discussed in Section 6. A sampling of some related work is presented in Section 8. Section 9 concludes the paper. A glossary of the notation used in this paper is given in Table 1.

2. System model

The system considered here consists of heterogeneous sets of sensors, applications, machines, and actuators. Each sensor produces data periodically at a certain rate, and the resulting data streams are input into applications. The applications process the data and send the output to other applications or to actuators.

In the types of environments under consideration, the domain experts (e.g., military officers, medical doctors) develop the code for the applications and the interactions among the applications. The applications and the data transfers between them are represented as a directed acyclic graphs (*DAGs*). The $\mathcal{A}$ nodes in the DAG correspond to the applications and the arcs between the nodes represent the data transfers between applications. Fig. 1 shows a DAG representation of the applications and the data transfers, along with the sensors and actuators in the system. Let

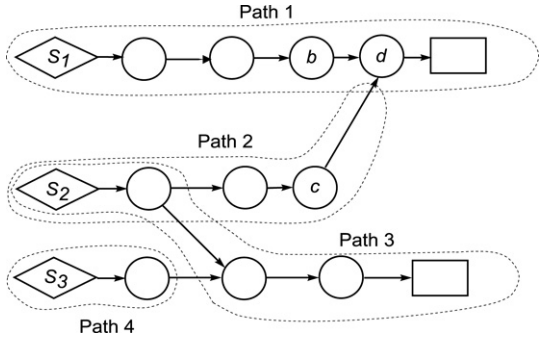

Fig. 1. The DAG model for the applications and data transfers. The dashed lines enclose each path formed by the applications. Diamonds represent sensors, circles represent applications, and rectangles represent actuators.

a_i be the i-th application in $\mathcal{A}$ (numbered arbitrarily). For our study, the DAGs are based on information provided to us about actual DAGs used by the Navy HiPer-D project at the Naval Surface Warfare Center, Dahlgren Division.

It is assumed that, for a given data set, the data-dependent applications use "greedy synchronization" to schedule their executions. Specifically, a successor application starts processing a data set as soon as it is received. It is also assumed that an application starts transferring data only after it has completed processing the current input.

We assume that all multiple input applications are *discriminating applications* [2]. Such an application does not produce an output unless a designated input stream (called the *trigger input*) supplies a new data set. The output is produced at the rate of the trigger input.

The figure also shows a number of paths (enclosed by dashed lines) formed by the applications. A *path* is a chain of producer-consumer pairs that starts at a sensor (the *driving sensor*) and ends at an actuator (if it is a *trigger path*) or at a multiple-input application (if it is an *update path*). In the context of Fig. 1, path 1 is a trigger path, and path 2 is an update path. In a real system, application d could be a missile firing program that produces an order to fire. It needs target coordinates from application b in path 1, and an updated map of the terrain from application c in path 2. Assume that application d must respond to any output from b, but must not issue fire orders if it receives an output from c alone; such an output is used only to update an internal database. So while d is a multiple input application, the rate at which it produces data is equal to the rate at which the trigger input, application b, produces data. That rate, in turn, equals the rate at which the driving sensor, S_1, produces data. We assume that as a part of the problem specification, we know the path to which each application belongs, and the corresponding driving sensor. Note that, in the above example, the system must have an initial map of the terrain to work appropriately. That is, before the system is started, a map is provided to application d. Of course such a map can be refined later by the update path.

Let $\mathcal{P}$ be the set of paths formed by applications in $\mathcal{A}$. Let $\mathcal{P}_k$ be the list of applications that comprise the k-th path. Let $|\mathcal{P}_k|$ be the total number of applications in path $\mathcal{P}_k$. Then, $\alpha_{k,i}$ is the i-th application in $\mathcal{P}_k$, where $1 \leq i \leq |\mathcal{P}_k|$. The applications in $\mathcal{P}_k$ are numbered sequentially. An application may be present in multiple paths, i.e., it is possible that $\alpha_{k,i} = \alpha_{l,j} = a_h$. Similarly, a given data transfer may be a part of multiple paths. To ensure that multiple applications do not drive a single actuator, only one path ends at an actuator. However, multiple paths can begin at a sensor.

The sensors constitute the interface of the system to the external world for retrieving information. These sensors may be radars, sonars, cameras monitoring a conveyor belt, etc. A sensor is characterized by (a) the maximum rate at which it produces data sets, and (b) the amount of load a given data set presents to the applications that have to process the data set. Typically, different sensors have different rates at which they produce data sets. Also, the measure of load may be different for different sensors. Let σ_z be the z-th sensor in the set of sensors, $\boldsymbol{\sigma}$. In this study, the *load* measure, λ_z, is defined to be the number of objects to be tracked present in one data set from the sensor σ_z in its most recent output. The *system workload*, $\boldsymbol{\lambda}$, is a measure of the load values from all sensors, and is the vector, $\boldsymbol{\lambda} = [\lambda_1 \cdots \lambda_{|\sigma|}]^{\mathrm{T}}$. Let $\boldsymbol{\lambda}^{\mathrm{init}}$ be the initial value of $\boldsymbol{\lambda}$, and $\lambda_i^{\mathrm{init}}$ be the initial value of the i-th member of $\boldsymbol{\lambda}^{\mathrm{init}}$, i.e., $\boldsymbol{\lambda}^{\mathrm{init}} = \left[\lambda_1^{\mathrm{init}} \cdots \lambda_{|\sigma|}^{\mathrm{init}}\right]^{\mathrm{T}}$.

The system also contains a set $\mathcal{M}$ of heterogeneous machines. Given that recent military programs specifically target reducing the cost of design, acquisition and upgrade of systems through the use of commercial-off-the-shelf equipment, we have assumed that the machines in our system have commercial-off-the-shelf operating systems. Specifically, we assume that when multiple applications are computed on a machine, the operating system uses a round-robin scheduling policy to allocate the CPU to the applications. Similarly, when there are multiple data transfers originating at a machine, we assume that the operating system uses a round-robin scheduling policy to allocate the network link to the data transfers. Section 3 develops expressions for the computation and communication times using the above assumptions.

All machines are connected by heterogeneous dedicated full-duplex communication links to a crossbar switch. The sensors, as well as actuators, are connected by half-duplex connections to the same crossbar switch. These assumptions reflect the shipboard computing environment for the High Performance Distributed Computing Program mentioned earlier.

Several QoS constraints, e.g., maximum end-to-end latency, inter-processing time, and throughput, have been discussed in [23]. In this research, we will investigate the maximum end-to-end latency and throughput constraints, as they are most pertinent to the problem domain. The *maximum end-to-end latency* constraint states that, for a given path $\mathcal{P}_k$, the time taken between the instant the driving sensor for the path outputs a data set to the instant the actuator or the discriminating application fed by the path receives the result of the computation on that data set must be no greater than a given value, $L_k^{\max}$.

The *minimum throughput constraint* states that the computation or communication time of any application in $\mathcal{P}_k$ is required to be no larger than the reciprocal of the output data rate of the driving sensor for $\mathcal{P}_k$. For application a_i in $\mathcal{P}_k$, let $R(a_i)$ be set to the output data rate of the driving sensor for $\mathcal{P}_k$.

3. Computation and communication models

Computation model: This part summarizes the computation model for this system as originally described in [2]. For an application, the estimated time to compute a given data set depends on the load presented by the data set, and the machine executing the application. Let $C_{ij}(\boldsymbol{\lambda})$ be the *estimated time to compute* for application a_i on m_j for a given workload (generated by $\boldsymbol{\lambda}$) when a_i is the only application executing on m_j. We assume that $C_{ij}(\boldsymbol{\lambda})$ is a function known for all i, j, and $\boldsymbol{\lambda}$ (e.g., estimated from analysis and experimentation).

We account for the effect of multitasking on the computation time of an application by making the simplifying worst-case assumption that all applications mapped to a machine are processing data continuously. Let N_j be the number of applications executing on machine m_j. Let $T_{ij}^c(\boldsymbol{\lambda})$ be the computation time for a_i on machine m_j when a_i shares this machine with other applications. If the overhead due to context switching is ignored, $T_{ij}^c(\boldsymbol{\lambda})$ will be $N_j \times C_{ij}(\boldsymbol{\lambda})$.

However, the overhead in computation time introduced by context switching may not be trivial in a round-robin scheduler. Such an overhead depends on the estimated-time-to-compute value of the application on the machine, and the number of applications being executed on the machine. Let $O_{ij}^{cs}(\boldsymbol{\lambda})$ be the context switching overhead incurred when application a_i executes a given workload on machine m_j. Let T_j^{cs} be the time m_j needs for switching the execution from one application to another. Let T_j^q be the size (quantum) of the time slice given by m_j to each application in the round-robin scheduling policy used on m_j. Then,

$$O_{ij}^{cs}(\boldsymbol{\lambda}) = \frac{C_{ij}(\boldsymbol{\lambda}) \times N_j}{T_j^q} \times T_j^{cs} \times u(N_j - 2), \tag{1}$$

where $u(N_j - 2)$ is a unit step function.[2] Given this,

$$T_{ij}^c(\boldsymbol{\lambda}) = C_{ij}(\boldsymbol{\lambda}) \times N_j \left(1 + \frac{T_j^{cs}}{T_j^q} \times u(N_j - 2)\right). \tag{2}$$

Communication model: In this part we develop an expression for the time needed to transfer the output data from a source application to a destination application at a given load. This formulation was originally given in [2]. Let $M_{ij}(\boldsymbol{\lambda})$ be the size of the message data sent from application a_i to a destination application a_j at the given load. Let $\mu(a_i)$ be the machine on which a_i is mapped. Let $T_{ij}^t(\boldsymbol{\lambda})$ be the transfer time, i.e., the time to send data from application a_i to application a_j at the given load.

A data packet is queued twice enroute from the source machine to the destination machine. First, the data packet is queued in the output buffer of the source machine, where it waits to use the communication link from the source machine to the switch. The second time, the data packet is queued in the output port of the switch, where it waits to use the communication link from the switch to the destination machine. The switch has a separate output port for each machine in the system.

We model the queuing delay at the sending machine by making the simplifying worst-case assumption that the bandwidth of the link from the sending machine to the switch is shared equally among all data transfers originating at the sending machine. This will underestimate the bandwidth available for each transfer because it assumes that all of the other transfers are always being performed. Similarly, we model the queuing delay at the switch by assuming that the bandwidth of the link from the switch to the destination machine is equally divided among all data transfers originating at the switch and destined for the destination machine. Let $B(\mu(a_i), \text{swt})$ be the bandwidth (in bytes per unit time) of the communication link between $\mu(a_i)$ and the switch, and $B(\text{swt}, \mu(a_j))$ be the bandwidth (in bytes per unit time) of the communication link between the switch and $\mu(a_j)$. The abbreviation "swt" stands for "switch". Let $N^{\text{ct}}(\mu(a_i), \text{swt})$ be the number of data transfers using the communication link from $\mu(a_i)$ to the switch. The superscript "ct" stands for "contention". Let $N^{\text{ct}}(\text{swt}, \mu(a_j))$ be the number of data transfers using the communication link from the switch to $\mu(a_j)$. Then, $T_{ij}^t(\boldsymbol{\lambda})$, the time to transfer the output data from application a_i to a_j, is given by:

$$T_{ij}^t(\boldsymbol{\lambda}) = M_{ij}(\boldsymbol{\lambda}) \times \left(\frac{N^{\text{ct}}(\mu(a_i), \text{swt})}{B(\mu(a_i), \text{swt})} + \frac{N^{\text{ct}}(\text{swt}, \mu(a_j))}{B(\text{swt}, \mu(a_j))}\right). \tag{3}$$

Eq. (3) also can accommodate the situations when a sensor communicates with the first application in a path, or when the last application in a path communicates with an actuator. The driving sensor for $\mathcal{P}_k$ can be treated as a "pseudo-application" that has a zero computation time and is already mapped to an imaginary machine, and, as such, can be denoted by $\alpha_{k,0}$. Similarly, the actuator receiving data from $\mathcal{P}_k$ can also be treated as a pseudo-application with a zero computation time, and will be denoted by $\alpha_{k,|\mathcal{P}_k|+1}$. $N^{\text{ct}}(\mu(\alpha_{k,0}), \text{swt})$ and $N^{\text{ct}}(\text{swt}, \alpha_{k,|\mathcal{P}_k|+1})$ both are 1. When $\mu(a_i) = \mu(a_j)$, $T_{ij}^t(\boldsymbol{\lambda}) = 0$.

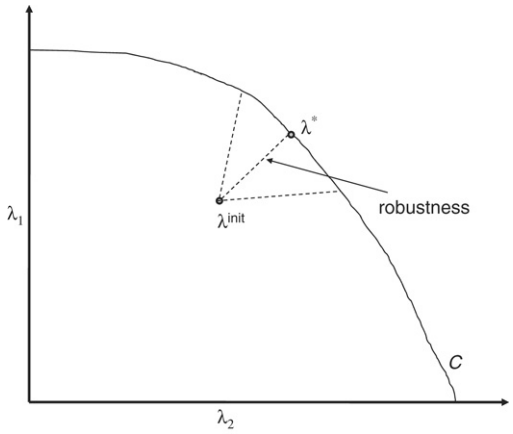

Fig. 2. Some possible directions of increase for the system load $\boldsymbol{\lambda}$, and the direction of the smallest increase to reach C. The region enclosed by the axes and the curve C gives the feasible values of $\boldsymbol{\lambda}$.

4. Performance goal

In this section we quantify the robustness of a mapping. Without loss of generality, to simplify the presentation, we assume that $\boldsymbol{\lambda}$ is a continuous variable, and that computation and communication times are continuous functions of $\boldsymbol{\lambda}$. The change in $\boldsymbol{\lambda}$ can occur in different "directions" depending on the relative changes in the individual components of $\boldsymbol{\lambda}$. For example, $\boldsymbol{\lambda}$ might change so that all components of $\boldsymbol{\lambda}$ increase in proportion to their initial values. In another case, only one component of $\boldsymbol{\lambda}$ may increase while all other components remain fixed. Fig. 2 illustrates some possible directions of increase in $\boldsymbol{\lambda}$. In Fig. 2, $\boldsymbol{\lambda}^{\text{init}} \in \mathbf{R}^2$ is the initial value of the system load. The region enclosed by the axes and the curve C gives the feasible values of $\boldsymbol{\lambda}$, i.e., all those values for which the system does not violate a QoS constraint. The element of C marked as $\boldsymbol{\lambda}^*$ has the feature that the Euclidean distance, $\|\boldsymbol{\lambda}^* - \boldsymbol{\lambda}^{\text{init}}\|$, from $\boldsymbol{\lambda}^{\text{init}}$ to $\boldsymbol{\lambda}^*$ is the smallest over all such distances from $\boldsymbol{\lambda}^{\text{init}}$ to a point on C. An important interpretation of $\boldsymbol{\lambda}^*$ is that the value $\|\boldsymbol{\lambda}^* - \boldsymbol{\lambda}^{\text{init}}\|$ gives the largest Euclidean distance that the variable $\boldsymbol{\lambda}$ can move in *any* direction from an initial value of $\boldsymbol{\lambda}^{\text{init}}$ without incurring a QoS violation [3]. *We define* $\Delta\boldsymbol{\Lambda} = \|\boldsymbol{\lambda}^* - \boldsymbol{\lambda}^{\text{init}}\|$ *to be the robustness of a mapping, against the system workload, with respect to satisfying the QoS constraints.*

We now give a conceptual way of determining $\Delta\boldsymbol{\Lambda}$. Assume application a_i is mapped to machine m_j. Let $\mathcal{D}(a_i)$ be the set of successor applications of a_i. Let $\mathcal{L}_i^{\text{T}}$ be the boundary of the set of all those $\boldsymbol{\lambda}$ values at which application a_i does not violate its throughput constraint, i.e.,

$$\mathcal{L}_i^{\text{T}} = \{\boldsymbol{\lambda} : T_{ij}^c(\boldsymbol{\lambda}) = 1/R(a_i)\} \bigcup \{\boldsymbol{\lambda} : \forall a_p \in \mathcal{D}(a_i), T_{ip}^t(\boldsymbol{\lambda}) = 1/R(a_i)\}. \tag{4}$$

The "T" in the superscript denotes "throughput". Let $\mathcal{L}^{\text{T}}$ be the boundary of the set of $\boldsymbol{\lambda}$ values at which no application violate its throughput constraint, i.e.,

$$\mathcal{L}^{\text{T}} = \bigcup_{a_i \in \mathcal{A}} \mathcal{L}_i^{\text{T}}. \tag{5}$$

[2] $u(N_j - 2) = 1$ if $N_j \geq 2$, else $u(N_j - 2) = 0$.

Similarly, let $\mathcal{L}_k^{\mathrm{L}}$ be the boundary of the set of those $\boldsymbol{\lambda}$ values at which path $\mathcal{P}_k$ does not violate its latency constraint, i.e.,

$$\mathcal{L}_k^{\mathrm{L}} = \left\{ \boldsymbol{\lambda} : \sum_{\substack{i:a_i \in \mathcal{P}_k, p:a_p \in \mathcal{P}_k \\ a_p \in \mathcal{D}(a_i)}} T_{ij}^c(\boldsymbol{\lambda}) + T_{ip}^t(\boldsymbol{\lambda}) = L_k^{\max} \right\}. \quad (6)$$

The "L" in the superscript denotes "latency". Let $\mathcal{L}^{\mathrm{L}}$ be the boundary of the set of $\boldsymbol{\lambda}$ values at which no path violates its latency constraint, i.e.,

$$\mathcal{L}^{\mathrm{L}} = \bigcup_{\mathcal{P}_k \in \mathcal{P}} (\mathcal{L}_k^{\mathrm{L}}). \quad (7)$$

Finally, let $\mathcal{L}$ be the set of $\boldsymbol{\lambda}$ given by $\mathcal{L}^{\mathrm{T}} \bigcup \mathcal{L}^{\mathrm{L}}$. One can then determine $\Delta\boldsymbol{\Lambda}$ by determining the smallest value of $\|\boldsymbol{\lambda} - \boldsymbol{\lambda}^{\mathrm{init}}\|$ for some $\boldsymbol{\lambda} \in \mathcal{L}$. That is,

$$\Delta\boldsymbol{\Lambda} = \min_{\boldsymbol{\lambda} \in \mathcal{L}} \|\boldsymbol{\lambda} - \boldsymbol{\lambda}^{\mathrm{init}}\|. \quad (8)$$

For later use, let $\Delta\boldsymbol{\Lambda}^{\mathrm{T}} = \min_{\boldsymbol{\lambda} \in \mathcal{L}^{\mathrm{T}}} \|\boldsymbol{\lambda} - \boldsymbol{\lambda}^{\mathrm{init}}\|$, and $\Delta\boldsymbol{\Lambda}^{\mathrm{L}} = \min_{\boldsymbol{\lambda} \in \mathcal{L}^{\mathrm{L}}} \|\boldsymbol{\lambda} - \boldsymbol{\lambda}^{\mathrm{init}}\|$.

We assume that the optimization problem given in the $\Delta\boldsymbol{\Lambda}$ equation given above can be solved to find the global minimum. An optimization problem of the form $x^* = \mathrm{argmin}_x f(x)$, subject to the constraint $g(x) = 0$, where $f(x)$ and $g(x)$ are convex and linear functions, respectively, can be solved easily to give the global minimum [9]. Because all norms are convex functions, the optimization problem posed in the $\Delta\boldsymbol{\Lambda}$ equation given above reduces to a convex optimization problem if $T_{ij}^c(\boldsymbol{\lambda})$ and $T_{ip}^t(\boldsymbol{\lambda})$ are linear functions, as they are in this work. If $T_{ij}^c(\boldsymbol{\lambda})$ and $T_{ip}^t(\boldsymbol{\lambda})$ functions are not linear, then it is assumed that known heuristic techniques could be used to find near-optimal solutions.

5. Heuristic descriptions

5.1. Greedy heuristics

In general, greedy techniques perform well in many situations, and have been well-studied (e.g., [16]). Our implementations of these heuristics, described later, use the $\Delta\boldsymbol{\Lambda}$ value as the performance metric. However, the procedure given in Section 4 for calculating $\Delta\boldsymbol{\Lambda}$ assumes that a complete mapping of all applications is known. During the course of the execution of the heuristics, not all applications are mapped. In these cases, the heuristics make the following unrealistic assumptions for the purpose of calculating an "intermediate" $\Delta\boldsymbol{\Lambda}$: (a) Each such application a_i is mapped to the machine m_j where $C_{ij}(\boldsymbol{\lambda}^{\mathrm{init}})$ is smallest over all machines, and (b) a_i is using 100% of that machine (i.e., there is no penalty for sharing). Similarly for the communications, it is assumed that each data transfer between two unmapped applications occurs over the highest speed communication link, and that the link is 100% utilized by the data transfer. With these assumptions, $\Delta\boldsymbol{\Lambda}$ is calculated and used in any step of a given heuristic.

Let $\Delta\boldsymbol{\Lambda}_{ij}^{\mathrm{T}}$ be the robustness of the assignment of a_i with respect to the throughput constraint, i.e., it is the largest increase in load in any direction from the initial value that does not cause a throughput violation for application a_i, either for the computation of a_i on machine m_j or for the communications from a_i to any of its successor applications. Then, $\Delta\boldsymbol{\Lambda}_{ij}^{\mathrm{T}} = \min_{\boldsymbol{\lambda} \in \mathcal{L}_i^{\mathrm{T}}} \|\boldsymbol{\lambda} - \boldsymbol{\lambda}^{\mathrm{init}}\|$. Similarly, let $\Delta\boldsymbol{\Lambda}_k^L$ be the robustness of the assignment of applications in $\mathcal{P}_k$ with respect to the latency constraint, i.e., it is the largest increase in load in any direction from the initial value that does not cause a latency violation for the path $\mathcal{P}_k$. It is given by $\min_{\boldsymbol{\lambda} \in \mathcal{L}_k^{\mathrm{L}}} \|\boldsymbol{\lambda} - \boldsymbol{\lambda}^{\mathrm{init}}\|$.

Most Critical Task First Heuristic. The *Most Critical Task First* (*MCTF*) heuristic (for throughput constrained systems) makes one application to machine assignment in each iteration. Each iteration can be split into two phases. Let $\Delta\boldsymbol{\Lambda}^*(a_i, m_j)$ be the value of $\Delta\boldsymbol{\Lambda}$ *if* application a_i is mapped on m_j. Similarly, let $\Delta\boldsymbol{\Lambda}^{\mathrm{T}^*}(a_i, m_j)$ be the value of $\Delta\boldsymbol{\Lambda}_{ij}^{\mathrm{T}}$ *if* application a_i is mapped on m_j. Let

$$G(a_i) = \underset{m_k \in \mathcal{M}}{\mathrm{argmax}}(\Delta\boldsymbol{\Lambda}^*(a_i, m_k)), \quad (9)$$

and

$$H(a_i) = \underset{m_k \in G(a_i)}{\mathrm{argmax}}(\Delta\boldsymbol{\Lambda}^{\mathrm{T}^*}(a_i, m_k)). \quad (10)$$

The function $\mathrm{argmax}_x f(x)$ returns the value of x that maximizes the function $f(x)$. If there are multiple values of x that maximize $f(x)$, then $\mathrm{argmax}_x f(x)$ returns the set of all those values.

In the first phase, each unmapped application a_i is paired with its "best" machine m_j such that $m_j = G(a_i)$ if $G(a_i)$ is a unique machine. Otherwise, the algorithm calculates $H(a_i)$ to select that machine from $G(a_i)$ that results in the largest robustness based on throughput constraints only. If $H(a_i)$ itself is not a unique machine, then one machine is randomly selected from $H(a_i)$. Let such a machine be denoted as $m_j = \mathrm{rand}(H(a_i))$. The first phase does not make an application to machine assignment; it only establishes application-machine pairs (a_i, m_j) for all unmapped applications a_i.

The second phase makes an application to machine assignment by selecting one of the (a_i, m_j) pairs produced by the first phase. This selection is made by determining the most "critical" application (the criterion for this is explained later). The method used to determine this assignment in the first iteration is totally different from that used in the subsequent iterations.

The special first iteration does the following. (1) Determine the single application that will result in the worst robustness when it alone is mapped to system. (2) Assign it to its best machine. (3) Let this assignment be the first one, so that the later assignments can "work around" this assignment. Mathematically, the special first iteration determines the pair (a_x, m_y) such that

$$(a_x, m_y) = \underset{(a_i, m_j) \text{ pairs from the first phase}}{\mathrm{argmin}} (\Delta\boldsymbol{\Lambda}^*(a_i, m_j)). \quad (11)$$

The application a_x is then assigned to the machine m_y.

The criterion used to make the second phase application to machine assignment for iteration number 2 to $|\mathcal{A}|$ is different from that used in iteration 1, and is now explained. The intuitive goal is to determine the (a_i, m_j) pair, which if not selected, may cause the most future "damage," i.e., decrease in $\Delta\boldsymbol{\Lambda}$. Let $\mathcal{M}^{a_i}$ be the ordered list, $\langle m_1^{a_i}, m_2^{a_i}, \ldots, m_{|\mathcal{M}|}^{a_i} \rangle$, of machines such that $\Delta\boldsymbol{\Lambda}^*(a_i, m_x^{a_i}) \geq \Delta\boldsymbol{\Lambda}^*(a_i, m_y^{a_i})$ if $x < y$. Note that $m_1^{a_i}$ is the same as a_i's "best" machine, $\mathrm{argmax}_{m_k \in \mathcal{M}}(\Delta\boldsymbol{\Lambda}^*(a_i, m_k))$. Let v be an integer such that $2 \leq v \leq |\mathcal{M}|$, and let $r(a_i, v)$ be the percentage decrease in $\Delta\boldsymbol{\Lambda}^*(a_i, m_1^{a_i})$ if a_i is mapped on $m_v^{a_i}$ (its v-th best machine) instead of $m_1^{a_i}$, i.e.,

$$r(a_i, v) = \frac{\Delta\boldsymbol{\Lambda}^*(a_i, m_1^{a_i}) - \Delta\boldsymbol{\Lambda}^*(a_i, m_v^{a_i})}{\Delta\boldsymbol{\Lambda}^*(a_i, m_1^{a_i})}. \quad (12)$$

Additionally, let $T(a_i)$ be defined such that

$$T(a_i) = \frac{\Delta\boldsymbol{\Lambda}^{\mathrm{T}^*}(a_i, m_1^{a_i}) - \Delta\boldsymbol{\Lambda}^{\mathrm{T}^*}(a_i, m_2^{a_i})}{\Delta\boldsymbol{\Lambda}^{\mathrm{T}^*}(a_i, m_1^{a_i})}. \quad (13)$$

Then, in all iterations other than the first iteration, MCTF maps the *most critical* application, where the most critical application is found using the pseudo-code in Fig. 3. The technique shown in Fig. 3 builds on the idea of the Sufferage heuristic concept given in [19].

Experiments indicated that doing only the first iteration differently from the remaining iterations improved the robustness of the mapping. For example, in one experiment, when the first two

(1) **initialize**: $v = 2$; $\mathcal{F}$ = the set of (a_i, m_j) pairs from the first phase
(2) **for** $v = 2$ to $|\mathcal{M}|$
(3) **if** $\text{argmax}_{(a_i,m_j)\in\mathcal{F}}(r(a_i, v))$ is a unique pair (a_x, m_y)
(4) **return** (a_x, m_y)
(5) **else**
(6) $\mathcal{F}$ = the set of pairs returned by $\text{argmax}_{(a_i,m_j)\in\mathcal{F}}(r(a_i, v))$
(7) **end for**
/* program control reaches here only if no application, machine pair has been */
/* selected in Lines 1 to 7 above. $\mathcal{F}$ is now the set of (a_i, m_j) pairs*/
/* from the last execution of Line 6*/
(8) **if** $\text{argmax}_{(a_i,m_j)\in\mathcal{F}}(T(a_i))$ is a unique pair (a_x, m_y)
(9) **return** (a_x, m_y)
(10) **else**
(11) arbitrarily select and return an application, machine pair from the set of pairs given by $\text{argmax}_{(a_i,m_j)\in\mathcal{F}}(T(a_i))$

Fig. 3. Selecting the most critical application to map next given the set of (a_i, m_j) pairs from the first phase of MCTF.

(1) **do** until all applications are mapped
(2) **for** each unmapped application a_i, find the machine m_j such that $m_j = \text{argmax}_{m_k\in\mathcal{M}}(\Delta\boldsymbol{\Lambda}^*(a_i, m_k))$; resolve ties arbitrarily
(3) **if** (a_i, m_j) assignment violates a QoS constraint, this heuristic cannot find a mapping
(4) from the (a_i, m_j) pairs found above, select the pair (a_x, m_y) such that $\Delta\boldsymbol{\Lambda}^*(a_x, m_y) = \max_{(a_i,m_j)\text{ pairs}}(\Delta\boldsymbol{\Lambda}^*(a_i, m_j))$; resolve ties arbitrarily
(5) map a_x on m_y
(6) **end do**

Fig. 4. The XXS heuristic.

(1) **do** until all applications are mapped
(2) **for** each unmapped application a_i, find the machine $m_j = \text{rand}(H(a_i))$
(3) **if** (a_i, m_j) assignment violates a QoS constraint, this heuristic cannot find a mapping
(4) **if** first iteration /* special case for the first iteration */
(5) from (a_i, m_j) pairs found above, select the pair (a_x, m_y) such that $\Delta\boldsymbol{\Lambda}^*(a_x, m_y) = \min_{(a_i,m_j)\text{ pairs}}(\Delta\boldsymbol{\Lambda}^*(a_i, m_j))$; resolve ties arbitrarily
(6) **else** /* for all but the first iteration */
(7) from (a_i, m_j) pairs found above, select the pair (a_x, m_y) such that $\Delta\boldsymbol{\Lambda}^*(a_x, m_y) = \max_{(a_i,m_j)\text{ pairs}}(\Delta\boldsymbol{\Lambda}^*(a_i, m_j))$; resolve any ties by choosing the most critical application first (Figure 3)
(8) map a_x on m_y
(9) **end do**

Fig. 5. The TB XXS heuristic.

iterations were done differently, the robustness reduced to 92% of the case where only one iteration was done differently.

Max–Max Style Greedy Heuristics: Fig. 4 shows a Max–max style greedy heuristic, *XXS*, that was implemented in this research (for throughput-constrained systems) as one benchmark against which our proposed heuristics can be compared. Variants of the Max–max heuristic (whose general concept was first presented in [16]) have been seen to perform well, e.g., [6,10,19]. *Tie-Breaker Max–max Style* heuristic (*TB XXS*), shown in Fig. 5, has the same first phase as MCTF. In addition, it augments the second phase of XXS with two features borrowed from MCTF: (a) the special first iteration, and (b) the use of application criticality to resolve any ties in Line (7) of Fig. 5.

Max–Min Style Greedy Heuristics: A Max–min style heuristic, *XNS* (based on the general concept in [16]), was also implemented to serve as another benchmark against which our proposed heuristics can be compared. XNS is similar to the XXS heuristic except that in Line (4) of Fig. 4, "max" is replaced with "min". The *Tie-Breaker Max–min Style* heuristic, *TB XNS*, was also implemented, and is related to TB XXS in the same way XNS is related to XXS.

Most Critical Path First Heuristic: The *Most Critical Path First* (*MCPF*) heuristic explicitly considers the latency constraints of the paths in the system, and is designed to work well in latency-constrained systems. MCPF begins by ranking the paths in the order of the most "critical" path first (defined below). Then it uses a modified form of the MCTF heuristic to map applications on a path-by-path basis, iterating through the paths in a ranked order. The modified form of MCTF differs from MCTF in that the first iteration has been changed to be the same as the subsequent iterations.

The ranking procedure used by MCPF is now explained in detail. Let $\hat{\boldsymbol{\Lambda}}^{\text{L}}(\mathcal{P}_k)$ be the value of $\Delta\boldsymbol{\Lambda}_k^L$ assuming that each application a_i in $\mathcal{P}_k$ is mapped to the machine m_j such that $j = \text{argmin}_{j:m_j\in\mathcal{M}} C_{ij}(\boldsymbol{\lambda}^{\text{init}})$, and that a_i can use 100% of m_j. Similarly for the communications between the consecutive applications in $\mathcal{P}_k$, it is assumed that

```
(1)  initial population generation and evaluation
(2)  while (stopping criteria not met), perform
(3)      selection; crossover; mutation; evaluation
(4)  end while
(5)  output best solution
```

Fig. 6. General procedure for a genetic algorithm.

each data transfer between two unmapped applications occurs over the highest speed communication link, and that the link is 100% utilized by the data transfer. The heuristic ranks the paths in an ordered list $G = \langle \mathcal{P}_1^{\text{crit}}, \mathcal{P}_2^{\text{crit}}, \ldots, \mathcal{P}_{|\mathcal{P}|}^{\text{crit}} \rangle$ such that $\hat{\mathbf{\Lambda}}^{\text{L}}(\mathcal{P}_x^{\text{crit}}) \leq \hat{\mathbf{\Lambda}}^{\text{L}}(\mathcal{P}_y^{\text{crit}})$ if $x < y$.

For an arbitrary HC system, one is not expected to know if the system is latency-constrained or throughput-constrained, or neither. In that case, this research proposes running both MCTF (or TB XXS) and MCPF, and taking the better of the two mappings. One such approach is our *Duplex* heuristic that executes both MCTF (one of the best throughput-constrained heuristics) and MCPF (a latency-constrained heuristic), and then chooses the mapping that gives a higher $\Delta\mathbf{\Lambda}$.

5.2. Random search algorithms

Genetic Algorithm: Genetic algorithms (GAs) are a technique for searching large solution spaces using computational analogs of biological and evolutionary processes. For some previous work on using GAs in scheduling, please see [21].

In a GA, a solution is encoded as a "chromosome". The algorithm begins by encoding a set (*population*) of possible solutions as chromosomes. This population is iteratively altered by the GA until a stopping criterion is met (Fig. 6).

In the selection step, some chromosomes are removed and others are duplicated based on their "fitness" values. A chromosome's *fitness* is a measure of the quality of the solution represented by that chromosome. Selection is followed by the crossover step, where chromosomes are randomly paired and, with a probability called the *crossover rate*, components of the paired chromosomes are exchanged to produce *offspring*. In the third step, called mutation, small random changes are introduced in the chromosomes for further population diversity. The mutation step is performed with a probability called the *mutation rate*. One iteration of the **while** loop in Fig. 6 constitutes one *generation*.

In this study each chromosome is a complete mapping of all applications to machines in the suite. In particular, a chromosome is a vector of $|\mathcal{A}|$ integers. Each element of the vector is a gene, and there are as many genes as there are applications in the system. Each gene can have a value in the set $\{1, \ldots, |\mathcal{M}|\}$. The value of the ith gene is the machine assigned to application a_i. The mapping robustness served as the fitness value.

In particular, the fitness value of a chromosome is calculated by determining the robustness (i.e., the $\Delta\mathbf{\Lambda}$ value of Eq. (8)) for the mapping (assignment of applications to machines) represented by that chromosome. The population size P was set to 100. Preliminary experiments were run for population sizes of 50, 100, and 200. It was observed that increasing P from 50 to 100 produced a significant improvement in the quality of solution. However, an increase in P from 100 to 200 did not change solution quality significantly.

The initial population was generated randomly, except for a "seed" chromosome that was the mapping found by the Duplex heuristic. Initial testing showed that such a seeding of the initial population produced a better mapping than the unseeded case. The seed mapping was taken from the Duplex heuristic because it was found that the Duplex heuristic always produced the best mapping among all greedy heuristics. The GA also used *elitism* to ensure that the fittest chromosome of a given generation was passed on to the next generation without being altered by the genetic operators.

We used linear ranking selection [7], in which the selection probability of a chromosome is a linear function of the chromosome's rank within the population. After selection, uniform crossover [22] was used. Specifically a set of $P/2$ chromosome pairs was formed by randomly selecting chromosomes from the current population. Each pair was chosen for crossover with a 90% probability. For each chosen pair, every gene was exchanged between the two parents with a 50% probability to produce two offspring. For the pairs not selected for crossover, parents were copied forward as offspring. For the mutation operation, a chromosome was chosen for mutation with a 5% probability. Then two genes were selected at random within the chosen chromosome, and the gene values were swapped. The GA was terminated after 200 generations. Preliminary experiments were run for several different crossover and mutation rates and total generations, and values given above were found to give the best results.

Further details of the GA operators and parameters are given in [6]. For a given problem, the GA was executed eight times,[3] each with a different initial population (except for the seed) and the best solution was chosen.

Steady State Genetic Algorithm: A steady state GA is a variation on a GA that uses overlapping populations, i.e., in a given iteration of the GA, only at most a given percentage of the population is replaced (in our work, 25%) [24]. Instead of linear ranking selection, offspring replace the least fit members of the population.

Simulated Annealing: The *simulated annealing* (*SA*) technique is an iterative improvement process of local search to obtain the global optimum of some given function. The SA technique has proven to be quite effective in approximating global optima for many different NP-hard combinatorial problems. In this study, the *two phase simulated annealing* (*TPSA*) heuristic employed in [17] was used.

The TPSA technique starts with an initial temperature, a function to decrease temperature, and an initial mapping solution (using the same chromosome format as the GA). In this research, the initial temperature is selected to be 10,000 because the temperature had to be sufficiently large at the start of the SA method such that the mapping solution does not converge quickly and fall into an early local minimum. The temperature is decreased at each iteration to 99% of the previous iteration's temperature. Preliminary experiments were run for different temperature decreasing schemes and stopping criteria and the values given in this study were found to give the best results. The mapping generated by Duplex is used as the initial solution. Preliminary experiments were run to decide whether seeding or not seeding the initial solution for SA was necessary and seeding the SA heuristic was found to give better results. In the preliminary experiments using the environment discussed in this paper, several different versions of SA-based heuristic were tested. In these experiments the basic SA (single phase with random initial solution) improved the Duplex solution by 5.5%, while the seeded SA, single phase, improved the Duplex solution by 15.6%. Therefore, it was concluded that seeding the SA heuristic with the solution found by Duplex would be best.

The heuristic runs until it meets one of the stopping criteria. The stopping criteria are when the temperature goes below 10^{-20} or when the current solution is unchanged for 150 iterations (from the stopping criterion described in [10]). At each iteration, the

[3] The number eight was chosen based on one of the author's prior experience with the "best of n" approach.

current solution is changed to make a new solution, and the new mapping is compared to the current mapping. If the new solution has a better robustness, then the new solution is chosen to be the current solution. When the new solution is worse than the current solution, it is probabilistically chosen to be the current solution. The uphill probability is determined using $e^{\frac{-|\text{value}_{\text{new}} - \text{value}_{\text{current}}|}{\text{temperature}}}$. It decides the probability of going uphill, e.g., if it is 0.2, then there is a 20% chance that the new solution will be chosen even though it is worse than the current solution.

When generating a new solution from the current solution, two methods are used. The first method randomly chooses a task and maps it onto a randomly chosen machine (*mutation*). The second method randomly chooses two tasks and swaps their machine assignments (*swap*). In the TPSA heuristic, there are two phases. In the first phase, the mutation method is used for the first 1,500 iterations or until the solution does not change for 100 iterations. Then, in the second phase, the swap method is used until the solution is not changed for 150 iterations or until the temperature is zero (i.e., 10^{-20}). For each trial, the heuristic is run ten times and the mapping with the best solution is selected. The intuition behind having two TPSA phases is that after some number of mutations, a near optimal number of tasks per machine will be found (reduced solution space). Then swapping two tasks will maintain the number of tasks on two machines (or on a machine) while trying to search for a better solution. Preliminary experiments were run for several different SA-based methods and the TPSA scheme was found to give better results.

The intuition behind using the swap operator in the second phase of the two phase SA (TPSA) is that in the first phase the SA heuristic is reducing the search space and in the second phase the swap method tries to find a better solution in this reduced space. But since the swap method is done in the second phase, it may already be difficult (if not impossible) to change the current solution to a worse solution. The swap method is a way to jump to another solution when the temperature is low and it is difficult to climb up by just mutating once. The swap method can be thought of as two mutations while maintaining a constraint (number of tasks on the machines do not change).

In the same preliminary experiments conducted above, the seeded TPSA heuristic improved the Duplex solution by 39% (compared to 5.5% and 15.6% for single phase unseeded and single phase seeded, respectively). The unseeded TPSA performed comparably to the seeded SA. Because the seeded TPSA's improvement was the largest, this scheme was chosen for the experiments in this paper.

An Upper Bound: The best way to evaluate the solution produced by a heuristic is to compare it with an optimal solution. Given that finding an optimal solution is NP-hard, we resorted to deriving a loose upper bound on the optimal solution, and compared our solutions to the upper bound. This upper bound is equal to the $\Delta\boldsymbol{\Lambda}$ for a system where the following assumptions hold: (a) the communication times are zero for all applications, (b) each application a_i is mapped on the machine m_j where $\Delta\boldsymbol{\Lambda}_{ij}^{\mathrm{T}}$ is maximum over all machines, and (c) each application can use 100% of the machine where it is mapped. These assumptions are, in general, not physically realistic. An example of when this upper bound situation could occur is: (a) applications do not communicate with each other, (b) each application is mapped to its best machine, and (c) the set of applications is such that only one application is mapped on a given machine.

6. Simulation experiments and results

In this study, several sets of simulation experiments were conducted to evaluate and compare the heuristics. Experiments were performed for different values of $|\mathcal{A}|$ and $|\mathcal{M}|$, and for different types of HC environments.

For the experiments presented here, $C_{ij}(\boldsymbol{\lambda})$ functions were generated in the following manner. Let f_{ijz} be a linear function of λ_z, with a slope $s^{\mathrm{c}}(a_i, m_j, \lambda_z)$ and an intercept $i^{\mathrm{c}}(a_i, m_j, \lambda_z)$. Then $C_{ij}(\boldsymbol{\lambda})$ was set to be the sum of all f_{ijz} for which there is a route from the sensor σ_z to a_i. Similarly, for generating functions for $M_{ij}(\boldsymbol{\lambda})$, let g_{ijz} be a linear function of λ_z, with a slope $s^{\mathrm{n}}(a_i, a_j, \lambda_z)$ and an intercept $i^{\mathrm{n}}(a_i, a_j, \lambda_z)$. Then, $M_{ij}(\boldsymbol{\lambda})$ was set to be the sum of all g_{ijz} for which there is a route from the sensor σ_z to a_i. These assumptions reflect one particular type of HC system. *However, the model developed in this study does not assume or require* $C_{ij}(\boldsymbol{\lambda})$ *or* $\mathrm{M}(a_i, a_j, \boldsymbol{\lambda})$ *to be linear functions of* $\boldsymbol{\lambda}$. *Any non-decreasing function can be used.*

For applications (a_i, a_j), machine (m_j), and sensor (λ_z), the $s^{\mathrm{c}}(a_i, m_j, \lambda_z)$, $s^{\mathrm{n}}(a_i, a_j, \lambda_z)$, $i^{\mathrm{n}}(a_i, a_j, \lambda_z)$, and $i^{\mathrm{c}}(a_i, m_j, \lambda_z)$ values were generated by randomly sampling a Gamma distribution [5]. The mean was arbitrarily set to 10, the "task heterogeneity" was set to 0.7, and the "machine heterogeneity" was also set to 0.7 (heterogeneity is the standard deviation divided by the mean). The chosen values are characteristic of systems with high application and machine heterogeneities [5]. The parameters of the distributions that generated $C_{ij}(\boldsymbol{\lambda})$ and $M_{ij}(\boldsymbol{\lambda})$ functions were kept the same because communication times in the HiPer-D target HC system are of the same order as computation times.

The next major step in setting up the simulations was to derive reasonable values for the output rates for the sensors and the end-to-end latency constraints for different paths in the system. To that end, both the output rates and end-to-end latency constraints should be related to the values of $s^{\mathrm{c}}(a_i, m_j, \lambda_z)$, $i^{\mathrm{c}}(a_i, m_j, \lambda_z)$, $s^{\mathrm{n}}(a_i, a_j, \lambda_z)$, and $i^{\mathrm{n}}(a_i, a_j, \lambda_z)$ derived above. Moreover, the rates of the target system sensors vary widely based on the scenario. Therefore, the simulation setup should allow changes in the sensor rates. The reader is directed to [6] for the discussion that explains the process used in the simulation setup to do this. Here, two control parameters are briefly mentioned. The parameter $w_{\mathrm{THR}} > 0$ is a real number that can be adjusted empirically to "tighten" the throughput constraint. Similarly, the parameter $w_{\mathrm{LAT}} > 0$ is a real number that can be adjusted to change the difficulty of meeting the latency constraint. The smaller the value of any one of these parameters, the "tighter" the corresponding constraint.

An *experiment* is characterized by the set of system parameters (e.g., $|\mathcal{A}|$, $|\mathcal{M}|$, application heterogeneity, and machine heterogeneity) it investigates. Each experiment was repeated enough times such that the width of the 95% confidence interval would be only 10% of the mean value. The number of repetitions to achieve this level of precision was different for different experiments, the maximum being 90. Call each repetition of a given experiment a *trial*. For each new trial, a DAG with $|\mathcal{A}|$ nodes was randomly regenerated, and the values of $s^{\mathrm{c}}(a_i, m_j, \lambda_z)$, $i^{\mathrm{c}}(a_i, m_j, \lambda_z)$, $s^{\mathrm{n}}(a_i, a_j, \lambda_z)$, and $i^{\mathrm{n}}(a_i, a_j, \lambda_z)$ were regenerated from their respective distributions. The minimum and maximum out-degree values for the DAG generation were 1 and 2, respectively.[4]

The results for a selected set of representative experiments are shown in Fig. 7. The first bar for each heuristic, titled "$\Delta\boldsymbol{\Lambda}$", shows the normalized $\Delta\boldsymbol{\Lambda}$ value averaged for all those trials in which a particular heuristic successfully found a mapping. The normalized $\Delta\boldsymbol{\Lambda}$ for a particular heuristic is equal to $\Delta\boldsymbol{\Lambda}$ for the mapping found by that heuristic divided by the upper bound on $\Delta\boldsymbol{\Lambda}$ defined in Section 5. The second bar, titled, "$\delta\boldsymbol{\lambda}$", shows the normalized $\Delta\boldsymbol{\Lambda}$ averaged only for those trials in which all heuristics successfully found a mapping. These figures also show, in the third bar, the value of failure rate for each heuristic. The *failure rate* is the ratio of the number of trials in which the heuristic could not find a mapping

[4] The procedure for generating the DAGs used in this study is given in [6].

Table 2
The relative performance of Duplex, GA, SSGA, and SA with respect to failure rate and $\Delta\mathbf{\Lambda}$

	Duplex	GA	SSGA	SA
Failure rate (%)	2	0	1	0
95% confidence interval for $\Delta\mathbf{\Lambda}$	0.78–0.87	0.80–0.88	0.78–0.87	0.83–0.91

to the total number of trials. The interval shown at the tops of the first two bars is the 95% confidence interval.

Fig. 7 shows the relative performance of the heuristics for the indicated system parameters. The values of w_{LAT} and w_{THR} were adjusted empirically to give a tightly constrained system. (For a lightly constrained system, all heuristics performed similarly.) The figure shows that Duplex, GA, SSGA, and SA have very similar performances with respect to the *robustness metric*. However, SA and GA do differ in their performance with respect to the *failure rate metric*. Both SA and GA have a failure rate of zero while others do not (see Table 2). Because it is extremely important for a heuristic to have a low failure rate in the target military application for this study, SA and GA give the best performance. This study concludes that in the environments where the heuristic running time is an issue, one should use Duplex and not try to improve its results by using SA or GA. However, if the heuristic running time is not an issue, *and* Duplex fails to find a mapping, one should proceed to using either GA or SA employing the Duplex mapping as a seed.

TB XXS performs better than XXS by a factor of almost 1.5 for $\Delta\mathbf{\Lambda}$ and 2 for failure rate. Recall the two features that distinguish TB XXS from XXS: (a) the special first iteration and (b) the use of application criticality to resolve any ties between applications in the subsequent iterations. Due to the nature of the optimization criterion, ties between applications are common in Line (4) of Fig. 4. Because these ties are resolved arbitrarily in XXS, the heuristic makes many arbitrary decisions during its course of execution. For the same reason, the tie-breaking improves TB XNS's $\Delta\mathbf{\Lambda}$ and failure rate values with respect to those for XNS.

The heuristic running times reflect the complexity of algorithms. The value of $\Delta\mathbf{\Lambda}$ is calculated $|\mathcal{M}||\mathcal{A}|^2$ times in all greedy heuristics, except for MCPF. For the experiment in Fig. 7, $|\mathcal{M}||\mathcal{A}|^2$ is 15,000, and the heuristic running times are in the vicinity of 0.24 seconds per trial for all greedy heuristics except for MCPF. In MCPF, applications are mapped on a path by path basis, and the average number of calculations of $\Delta\mathbf{\Lambda}$ is $|\mathcal{P}| \times \left(|\mathcal{M}| \frac{|\mathcal{A}|}{|\mathcal{P}|}^2\right)$, hence the running time for MCPF is much smaller that those of other greedy heuristics. GA makes 20,100 evaluations of $\Delta\mathbf{\Lambda}$ (100 initially and then another 100 in each of the 200 iterations).

Extrapolating from the running times of MCTF, XXS, TB XXS, XNS, and TB XNS, the predicted running time for GA should be approximately 0.56 seconds, derived from MCTF time (for seeding GA) $+ \frac{20,100}{15,000} \times 0.24$. The actual time is larger (0.60 s, probably due to selection, crossover and mutation operations). Because SSGA makes about 25% as many evaluations as GA makes (because only 25% of the population is replaced in each iteration), its running time should be expected to be 0.33 s, derived from MCTF time (for seeding SSGA) $+ \frac{20\,100}{4\times 15\,000} \times 0.24$. This is consistent with the experiments. The advantage of the partial population replacement is that SSGA runs much faster than GA. However, the disadvantage is that SSGA did not always find a feasible resource allocation as opposed to GA that always found a feasible resource allocation.

This paper focuses on finding robust solutions. An important question is whether such robust solutions also have good system utility. For example, it may always be possible to find a robust solution at the expense of system performance. We did further experiments to evaluate our robust solutions for "system slack", and found that it was possible that our algorithms would worsen the slack value at the expense of a robustness. More interestingly, we found that a solution with very good slack and bad robustness could be modified to give almost the same slack but much better robustness. An example of this is presented in [3].

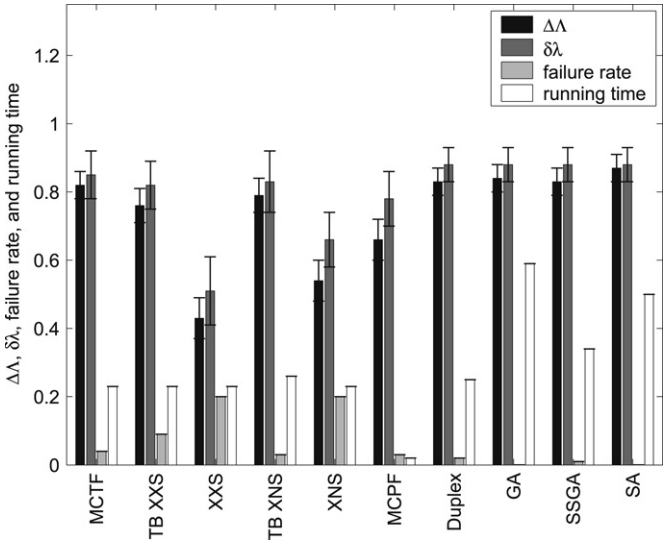

Fig. 7. The relative performance of heuristics for a system where $|\mathcal{M}| = 6$, $|\mathcal{A}| = 50$, $w_{\text{LAT}} = 8$, and $w_{\text{THR}} = 15$. Number of sensors = number of actuators = 7.

7. Future work

As one extension of this work, we would like to determine if we can replace GA or SA with lower cost algorithms to follow after Duplex has failed to find a mapping. Some options could be expanding the search around Duplex's mapping using, for example, a stochastic hill climber. We also could try randomizing Duplex with heuristic biased stochastic sampling, heuristic equivalency, and value biased stochastic sampling.

As another extension, we would like to explore trade-offs between choosing different norms for the robustness, e.g., the Manhattan distance or the infinity-norm distance instead of the Euclidean distance.

This work considers only finding a complete mapping that meets constraints or else the mapping heuristic has failed. Another future area to explore is considering partial mappings, where some application is not mapped due to the inability to find an assignment for that application to allow it to meet constraints. One possible approach is to modify the heuristics so they do not halt when a constraint violation for a given application cannot be met, but instead map as many applications as possible.

8. Related work

A number of papers in the literature have studied the issue of robustness in distributed computing systems (e.g., [2,8,13–15,18]). These studies are compared below with our paper.

The study in [2] uses a computing environment similar to the one in this paper; however, the robustness model in [2] is much simpler. Specifically, [2] assumes that $\boldsymbol{\lambda}$ changes so that all components of $\boldsymbol{\lambda}$ increase in proportion to their initial values. That is, if the output from a given sensor increases by x%, then the output from all sensors increases by x%. Given this assumption, for any two sensors σ_p and σ_q, $(\lambda_p - \lambda_p^{\text{init}})/\lambda_p^{\text{init}} = (\lambda_q - \lambda_q^{\text{init}})/\lambda_q^{\text{init}} = \Delta\lambda$. For this particular definition of an increase in the system workload, any function of the vector $\boldsymbol{\lambda}$ is in reality only a function of the

single scalar parameter, $\Delta\lambda$ (because $\boldsymbol{\lambda}^{\text{init}}$ is a constant vector). Our current research does not make this simplifying assumption; as a result, the approach taken in our current paper is much more general than that in [2].

Given an allocation of a set of communicating applications to a set of machines, the work in [8] investigates the robustness of the makespan against uncertainties in the estimated execution times of the applications. The paper discusses in detail the effect of these uncertainties on the value of makespan, and how to find more robust resource allocations. Based on the model and assumptions in [8], several theorems about the properties of robustness are proven. The robustness metric in [8] was formulated for errors in the estimation of application execution times; our measure is formulated for unpredictable increases in the system load. Additionally, the formulation in [8] assumes that the execution time for any application is at most k times the estimated value, where $k \geq 1$ is the same for all applications. In our work, no such constraint is assumed on the system workload.

The research in [13] considers a single-machine scheduling environment where the processing times of individual jobs are uncertain. The system performance is measured by the total flow time (i.e., the sum of *completion* times of all jobs). Given the probabilistic information about the processing time for each job, the authors determine the normal distribution that approximates the flow time associated with a given schedule. A given schedule's robustness is then given by 1 minus the risk of achieving substandard flow time performance. The risk value is calculated by using the approximate distribution of flow time.

The study in [14] explores slack-based techniques for producing robust resource allocations in a job–shop environment. The central idea is to provide each task with extra time (defined as slack) to execute so that some level of uncertainty can be absorbed without having to reallocate. The study uses slack as its measure of robustness.

The research in [15] introduces techniques to incorporate fault tolerance in scheduling approaches for real-time systems by the use of additional time to perform the system functions (e.g., to re-execute, or to execute a different version of, a faulty task). Their method guarantees that the real-time tasks will meet the deadlines under transient faults, by reserving sufficient additional time, or slack. Given a certain system slack and task model, the paper defines its measure of robustness to be the "fault tolerance capability" of a system (i.e., the number and frequency of faults it can tolerate). This measure of robustness is similar, in principle, to ours, but the models for applications and robustness are quite different.

The work in [18] develops a mathematical definition for the robustness of makespan against machine breakdowns in a job–shop environment. The authors assume a certain random distribution of the machine breakdowns and a certain rescheduling policy in the event of a breakdown. Given these assumptions, the robustness of a schedule is defined to be a weighted sum of the expected value of the makespan of the rescheduled system, M, and the expected value of the schedule delay (the difference between M and the original value of the makespan). Because the analytical determination of the schedule delay becomes very hard when more than one disruption is considered, the authors propose surrogate measures of robustness that are claimed to be strongly correlated with the expected value of M and the expected schedule delay.

The classes of distributed constraint satisfaction problems (DCSP) [25] and distributed constraint optimization problems (DCOP) [20] are similar to the problem discussed in this paper. DCOP is closer because like DCOP we not only ensure that all QoS constraints are satisfied but also that a certain performance measure (in our case, distance to the closest possible violation) is maximized. However, in our case, while the target system is distributed, the resource allocation algorithm is not itself distributed.

In summary, these examples show that robust resource allocation is a problem of interest. However, our work significantly differs from [2,8,13–15,18] in system modeling and the generality of its application.

9. Conclusions

Two distinct issues related to a resource allocation are investigated: its robustness and the failure rate of the heuristic used to determine the allocation. The system is expected to operate in an uncertain environment where the workload, i.e., the load presented by the set of sensors, is likely to increase unpredictably, possibly invalidating a resource allocation that was based on the initial workload estimate. The focus of this work is the design of a static heuristic that: (a) determines a *robust* resource allocation, i.e., a resource allocation that maximizes the allowable increase in workload until a run-time reallocation of resources is required to avoid a quality of service violation, and (b) has a very low *failure rate*. A failure occurs if no allocation is found that allows the system to meet its resource and quality of service constraints. This study proposes two heuristics that perform well with respect to the failure rates and the robustness towards unpredictable workload increases. These heuristics are, therefore, very desirable for systems where low failure rates can be a critical requirement and where unpredictable circumstances can lead to unknown increases in the system workload.

The heuristics were compared under a variety of simulated heterogeneous computing environments. Genetic Algorithm and Simulated Annealing performed the best, giving significantly lower failure rates, and no worse robustness compared to all other heuristics. Therefore Genetic Algorithm and Simulated Annealing are heuristics of choice for these heterogeneous computing environments, especially when it is critical to find a feasible mapping in tightly constrained systems where other heuristics discussed in this research may fail.

Acknowledgments

A preliminary version of portions of this material appeared in the 2004 International Conference on Parallel Processing (ICPP 2004) [4]. The authors thank Sameer Shivle and the anonymous reviewers for their valuable comments.

References

[1] S. Ali, T.D. Braun, H.J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, Characterizing Resource Allocation Heuristics for Heterogeneous Computing Systems, in: A.R. Hurson (Ed.), Parallel, Distributed, and Pervasive Computing, in: Advances in Computers, vol. 63, Elsevier Academic Press, San Diego, CA, 2005, pp. 91–128.

[2] S. Ali, J.-K. Kim, Y. Yu, S.B. Gundala, S. Gertphol, H.J. Siegel, A.A. Maciejewski, V. Prasanna, Greedy heuristics for resource allocation in dynamic distributed real-time heterogeneous computing systems, in: 2002 International Conference on Parallel and Distributed Processing Techniques and Applications, vol. II, June 2002, pp. 519–530.

[3] 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.

[4] S. Ali, A.A. Maciejewski, H.J. Siegel, J.-K. Kim, Robust resource allocation for sensor-actuator distributed computing systems, in: 2004 International Conference on Parallel Processing, ICPP 2004, August 2004, pp. 178–185.

[5] S. Ali, H.J. Siegel, M. Maheswaran, D. Hensgen, S. Sedigh-Ali, Representing task and machine heterogeneities for heterogeneous computing systems, Tamkang Journal of Science and Engineering 3 (3) (2000) 195–207. invited.

[6] S. Ali, Robust resource allocation in dynamic distributed heterogeneous computing systems, Ph.D. Thesis, School of Electrical and Computer Engineering, Purdue University, August 2003.

[7] J.E. Baker, Reducing bias and inefficiency in the selection algorithm, in: 2nd International Conference on Genetic Algorithms and Their Application, Lawrence Erlbaum Associates, Inc., 1987, pp. 14–21.
[8] L. Bölöni, D.C. Marinescu, Robust scheduling of metaprograms, Journal of Scheduling 5 (5) (2002) 395–412.
[9] S. Boyd, L. Vandenberghe, Convex optimization. Available at: http://www.stanford.edu/class/ee364/index.html.
[10] T.D. Braun, H.J. Siegel, N. Beck, L.L. Bölöni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D. Hensgen, R.F. Freund, 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.
[11] Naval Surface Warfare Center Report of the HiPer-D C&CA Working Group. http://www.nswc.navy.mil/hiperd/reports/thrust/, 1999.
[12] E.G. Coffman Jr., Computer and Job–Shop Scheduling Theory, John Wiley & Sons, New York, NY, 1976.
[13] R.L. Daniels, J.E. Carrillo, β-Robust scheduling for single-machine systems with uncertain processing times, IIE Transactions 29 (11) (1997) 977–985.
[14] A.J. Davenport, C. Gefflot, J.C. Beck, Slack-based techniques for robust schedules, in: 6th European Conference on Planning, ECP-2001, September 2001, pp. 7–18.
[15] S. Ghosh, Guaranteeing fault tolerance through scheduling in real-time systems, Ph.D. Thesis, Faculty of Arts and Sciences, Univ. of Pittsburgh, 1996.
[16] O.H. Ibarra, C.E. Kim, Heuristic algorithms for scheduling independent tasks on nonidentical processors, Journal of the ACM 24 (2) (1977) 280–289.
[17] J.-K. Kim, Resource management in heterogeneous computing systems: Continuously running applications, tasks with priorities and deadlines, and power constrained mobile devices, Ph.D. Thesis, School of Electrical and Computer Engineering, Purdue University, Aug. 2004.
[18] V.J. Leon, S.D. Wu, R.H. Storer, Robustness measures and robust scheduling for job shops, IEE Transactions 26 (5) (1994) 32–43.
[19] 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) 107–131.
[20] P.J. Modi, W.-M. Shen, M. Tambe, M. Yokoo, An asynchronous complete method for distributed constraint optimization, in: 2nd International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS 2003, July 2003, pp. 161–168.
[21] G. Syswerda, J. Palmucci, The application of genetic algorithms to resource scheduling, in: 4th International Conference on Genetic Algorithms, July 1991, pp. 502–508.
[22] G. Syswerda, Uniform crossover in genetic algorithms, in: 3rd International Conference on Genetic Algorithms, June 1989, pp. 2–9.
[23] L.R. Welch, P.V. Werme, B. Ravindran, L.A. Fontenot, M.W. Masters, D.W. Mills, B.A. Shirazi, Adaptive QoS and resource management using a-posteriori workload characterizations, in: 5th IEEE Real-Time Technology and Applications Symposium, RTAS'99, June 1999, pp. 266–275.
[24] D. Whitley, The GENITOR algorithm and selective pressure: Why rank- based allocation of reproductive trials is best, in: J. Schaffer (Ed.), 3rd International Conference on Genetic Algorithms, June 1989, pp. 116–121.
[25] M. Yokoo, E.H. Durfee, T. Ishida, K. Kuwabara, The distributed constraint satisfaction problem: formalization and algorithms, IEEE Transactions on Knowledge and Data Engineering 10 (5) (1998) 673–685.

Shoukat Ali is currently a senior engineer at Intel. Before that he was an assistant professor in the Electrical and Computer Engineering Department at the University of Missouri — Rolla, which he joined in August 2003. He received his M.S. and Ph.D. in Electrical and Computer Engineering in August 1999 and August 2003, respectively. He received his B.S. degree in Electrical Engineering from the University of Engineering and Technology, Lahore, Pakistan. His research interests are in resource allocation algorithms and computer architecture. In these fields, he has co-authored 7 journal papers, 17 conference papers, and 2 book chapters.

Jong-Kook Kim is currently an Assistant Professor at Korea University, Seoul Korea. He received his M.S. (2000) and Ph.D. (2004) degrees from Purdue University. He received his B.S. degree in electronic engineering from Korea University, Seoul, Korea in 1998. He has worked at Samsung SDS's IT R & D Center from 2005 to 2007. His research interests include heterogeneous distributed computing, efficient and reliable computing, computer architecture, resource management, evolutionary heuristics and energy-aware computing. He is a member of the IEEE, IEEE Computer Society, and ACM. For more information please visit jongkook.kim.googlepages.com.

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. For more information please visit www.engr.colostate.edu/~hj.

Anthony A. Maciejewski received the B.S.E.E., M.S., and Ph.D. degrees from Ohio State University in 1982, 1984, and 1987. From 1988 to 2001, he was a professor of Electrical and Computer Engineering at Purdue University, West Lafayette. He is currently the Department Head of Electrical and Computer Engineering at Colorado State University. He is a Fellow of the IEEE. A complete vita is available at: www.engr.colostate.edu/~aam.