J. Parallel Distrib. Comput. 73 (2013) 1705–1717

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

Robust static resource allocation of DAGs in a heterogeneous multicore system☆

Luis Diego Briceño [a,*], Jay Smith [a,c], Howard Jay Siegel [a,b], Anthony A. Maciejewski [a], Paul Maxwell [a,d], Russ Wakefield [b], Abdulla Al-Qawasmeh [a], Ron C. Chiang [a], Jiayin Li [a]

[a] Department of Electrical and Computer Engineering, Colorado State University, United States
[b] Department of Computer Science, Colorado State University, United States
[c] Lagrange Systems, United States
[d] United States Army, United States

HIGHLIGHTS

- Modeled a complex heterogeneous computing system that uses DAGs.
- Defined a robustness metric for this system.
- Designed heuristics to maximize this robustness metric.
- Evaluated and compared the performance of these heuristics in six scenarios.

ARTICLE INFO

Article history:
Received 15 October 2012
Received in revised form
30 July 2013
Accepted 8 August 2013
Available online 27 August 2013

Keywords:
Directed acyclical graph
Heterogeneous computing
Resource allocation

ABSTRACT

In this study, we consider an environment composed of a heterogeneous cluster of multicore-based machines used to analyze satellite data. The workload involves large data sets and is subject to a deadline constraint. Multiple applications, each represented by a directed acyclic graph (DAG), are allocated to a dedicated heterogeneous distributed computing system. Each vertex in the DAG represents a task that needs to be executed and task execution times vary substantially across machines. The goal of this research is to assign the tasks in applications to a heterogeneous multicore-based parallel system in such a way that all applications complete before a common deadline, and their completion times are robust against uncertainties in execution times. We define a measure that quantifies robustness in this environment. We design, compare, and evaluate five static resource allocation heuristics that attempt to maximize robustness. We consider six different scenarios with different ratios of computation versus communication, and loose and tight deadlines.

1. Introduction

In this study, we consider a *h*eterogeneous *c*omputing (*HC*) system based on multicore chips used to analyze satellite data. The data processing applications used in the analysis typically require computation on large data sets, and their execution may be subject to a completion deadline. Multiple applications, each represented as a *d*irected *a*cyclic *g*raph (*DAG*) of tasks, are to be assigned to an HC system for execution. Each vertex in the DAG represents tasks that need to be executed and task execution times vary substantially across machines. The goal of this study is to assign tasks to processors in such a way that all applications complete before a common deadline, and the application completion times are robust against uncertainties in task execution times. The robustness metric in this environment can be viewed as a quality of service guarantee, i.e., we can guarantee that all tasks' execution time can increase by some factor without the need to fix the schedule. We define a measure of robustness in this context, and we design, compare, and evaluate five static resource allocation heuristics that attempt to maximize robustness and test them under a variety of situations. We evaluate the performance of these heuristics in six different scenarios.

The simulation environment used to compare and evaluate these static resource allocation heuristics is motivated by similar systems in use at DigitalGlobe [10] and the National Center for Atmospheric Research [21] (*NCAR*). In these systems, data from a

☆ This research was supported by the NSF under grants CNS-0615170 and CNS-0905399, and by the Colorado State University George T. Abell Endowment.
* Corresponding author.
E-mail addresses: luis.d.briceno@gmail.com, luis.d.briceno.guerrero@intel.com (L.D. Briceño).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jpdc.2013.08.007

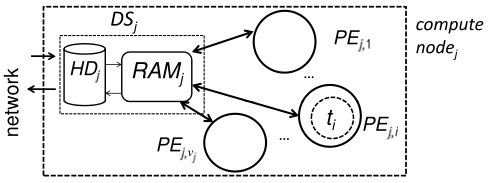


Fig. 1. The composition of compute node j.

satellite is received and distributed to storage units in the satellite data processing HC system, where there are (a) heterogeneous hard drive (*HD*) access rates, (b) different computational capabilities across compute nodes, and (c) different data set sizes. This satellite data processing system has the following characteristics: (a) the initial allocation of satellite data to HDs is determined by the resource allocation heuristic, (b) there is limited RAM available at each compute node, (c) data items have to be explicitly staged to and removed from RAM, (d) some tasks can be executed using parallelization, and (e) transfer times between HD and RAM must be taken into account. The simulation environment models this HC system and the applications. In this environment and other similar environments, e.g., the Adaptive and Reflective Middleware Systems shipboard environment [22], we need a static offline resource allocation that can handle a large degree of uncertainty without the need to do a run-time reallocation of resources to correct the schedule.

The reason we assume the computation time can vary and not the communication time is based on our assumed environment. We consider the problem domain where applications are DAGs, and the execution time of each task in the application is data dependent. The amount of data that needs to be transferred between tasks may be independent of the content of the data set being processed by the task, e.g., the temperatures calculated for a given list of coordinates. Also, for the systems at NCAR and Digital Globe, the communications are within a single parallel machine, and the transfer time between any two nodes remains a fixed function of the data set size; we allow the bandwidth to be a function of the pair of compute nodes communicating.

Each application requires only a subset of the total collection of data that is downloaded from the satellite. Resource allocation in this environment requires both selecting a location within the system to store each satellite data item and mapping tasks to compute nodes for execution; the mapping includes a decision about using data parallelism and multiple cores for each task. All applications and their required satellite data items are known prior to the satellite collecting the data, so this is an instance of a static resource allocation problem [1,6]. The general mapping problem is NP-complete [8,11,14]; therefore, heuristics are required to obtain a near-optimal allocation in a reasonable time.

Our contributions are (a) a model and simulation of a complex multicore-based data processing environment that executes data intensive applications, (b) a robustness metric for this environment, and (c) static resource allocation heuristics to maximize robustness using this metric.

In the next section, we will describe the problem statement. Five heuristics are defined in Section 3. The related work is discussed in Section 4. Sections 5 and 6 provide the results and conclusions, respectively.

2. Problem statement

2.1. System model

This HC system is composed of $\boldsymbol{N}$ compute nodes, where each compute node j has dedicated storage ($\boldsymbol{DS_j}$)—composed of RAM_j and HD_j. Each compute node also has one to eight processing elements (*PEs*), where each PE in our model may correspond to a core in a real system. We make the simplifying assumption that each PE may execute only one task at a time, i.e., no multi-tasking. The PEs within a compute node are homogeneous, but are assumed to be heterogeneous across compute nodes. Each compute node j has $\boldsymbol{\nu_j}$ PEs where the xth PE is denoted by $\boldsymbol{PE_{j,x}}$ ($1 \leq x \leq \nu_j$), and the total number of PEs across all compute nodes is $\boldsymbol{M}$ ($M = \sum_{j=1}^{N} \nu_j$). The composition of a compute node j is shown in Fig. 1.

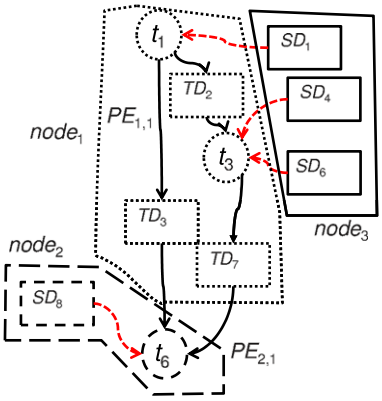


Fig. 2. An example diagram of a DAG mapped to three compute nodes. The PE shown in compute node 1 ($PE_{1,1}$) is executing t_1 that requires SD_1 from compute node 3. In this case, TD_3 and TD_7 on compute node 1 need to be transmitted to compute node 2 for t_6. The result of t_6 must be stored in an HD of the system, and the time to store the result must be considered when calculating the makespan.

The goal of resource allocations in this environment is to complete all applications before a common deadline ($\boldsymbol{\Delta}$). Let $\boldsymbol{app_k}$ be the kth application. Each app_k is divided into $\boldsymbol{T_k}$ tasks, represented by a DAG. In the DAG, entry tasks require only satellite data and exit tasks produce final results that must be stored on an HD. Without loss of generality, we assume that there is a single entry task and a single exit task for each DAG. Tasks may require both satellite data sets, denoted by $\boldsymbol{SD_i}$, and data sets produced by other tasks, denoted by $\boldsymbol{TD_j}$. Within a compute node α, a data set can be located in either RAM or the HD. The location of a data set (TD or SD) within a compute node α is denoted by $\boldsymbol{loc}_\alpha$, i.e., $loc_\alpha \in \{RAM_\alpha, HD_\alpha\}$. An example of a resource allocation of a DAG is shown in Fig. 2.

In each compute node, the RAM storage space is limited and may be unable to store simultaneously all of the data sets currently needed by tasks assigned to the PEs; however, each HD is assumed to be large enough to store all SDs and TDs assigned to it. If a TD in RAM is to be released, but is required later as an input to a task, then it must be copied to the HD. Because all SDs are initially stored on HDs, SDs do not need to be saved to the HD before being overwritten in RAM. The heuristics can generate the initial placement of satellite data among the compute nodes. The method used to place the data is to ignore the data transfer cost for the first task that uses the satellite data and place each satellite data set on the compute node that requires it first. We compared this to initially placing this data randomly and the heuristic approach always performed comparable or better.

For this study, we assume all the input data sets required by a task must be in local RAM before the task can start executing and must remain in RAM until its execution is finished. The storage space in RAM for the output of a task must be reserved locally before it begins execution. For a given compute node, that is the destination for a TD, if there is space currently available in the destination RAM then the required data sets are sent directly to that RAM. If there is no space currently available in the destination RAM, then the required data is sent to the HD of the destination node.

When a space is needed in RAM for a new TD or SD, we evaluate which data sets can be removed from RAM. We select for removal

data sets that are not currently being used and have the fewest remaining tasks that require them until we have freed the needed space. A task cannot be assigned to a node if the required space cannot be made available. If a TD that is removed is needed later, it will copied to the HD to avoid lost data.

In this model, all PEs on the same compute node share dedicated storage and network access. The time to access local RAM from a PE is assumed to be part of the estimated task execution time. HD access must be considered explicitly and is different for a read or a write.

The network topology used for this study is a *h*igh *p*erformance *p*arallel *i*nterface (*HiPPI*) crossbar switch. Each compute node may simultaneously transmit and receive one data set at a time, but may not broadcast. Possible contention transmitting and receiving data are handled by the resource allocation. The transfer rate of data from one compute node to another depends on the data's location in both the source and the destination, i.e., RAM or HD.

Because there are two locations where data may be stored and two compute nodes involved in the transfer, there are four cases of data transfer within this system. In the first case, we wish to transfer data from RAM on a source compute node to RAM on a destination compute node. This transfer is only limited by network bandwidth (the same for all compute nodes) because the bandwidth to RAM is always greater. In the second case, the transfer rate of data from the HD on the source compute node to RAM on the destination compute node is limited by the smaller of the network bandwidth and the read bandwidth of the source HD. In the third case, the transfer rate of data from RAM on a source compute node to HD on a destination compute node is limited to the smaller of the network bandwidth and the write bandwidth of the destination HD. In the fourth case, the transfer rate of data from an HD on the source compute node to the HD on the destination compute node is limited by the smaller of the network bandwidth, the read bandwidth from the source HD, and the write bandwidth to the HD on the destination compute node. If the destination node is the same as the source node the RAM to RAM and HD to HD transfer times are zero because there is no data movement.

The characteristics of typical tasks are from suites provided by the computing site (e.g., DigitalGlobe, NCAR). Thus, the estimated computation time of these tasks can be determined by historical information or experimentation. For each task $\boldsymbol{t_i}$, we assume that an *e*stimated *t*ime to *c*ompute on each compute node j has been provided, denoted by $\boldsymbol{ETC(i,j)}$, which is a common assumption in the literature (e.g., [5,9,12,15,16,26,33]). The goal of this study is to assign tasks to PEs so that unexpected increases in the estimated task computation times do not cause the total time required to complete all applications (*makespan*) to exceed Δ.

A subset of the tasks are designed to be decomposable for parallel execution within a single compute node. These decomposable tasks are grouped into "good" parallel tasks and "worse" parallel tasks, based on how amenable the task is to parallel processing. For both "good" and "worse" parallel tasks, a *divisor* value is used to scale the execution time depending on how many PEs a task is using. This *parallel* execution time is denoted by $\boldsymbol{ETC_{parallel}(i,j)} = \frac{ETC(i,j)}{divisor}$, and the divisor values we use in the simulations are shown in Table 1. In a real environment, these divisors can be estimated based on experiments or algorithm parallelization analysis. In this work, whenever a parallelizable task is assigned to a compute node all idle PEs were assigned to execute the task. In our simulation environment, the amount of RAM in the compute node is a constraint that can prevent all PEs from executing different tasks. Therefore, parallelizing tasks makes more efficient use of the PEs.

In this simulation environment, a mapping consists of allocating tasks to PEs in such a way that they obey precedence constraints (both execution and communication). The order in which tasks execute on a given PE does not change, nor does the order in which the communication is done.

Table 1
Divisor for parallel tasks.

Types of parallelism	Number of PEs in use							
	1	2	3	4	5	6	7	8
Divisor for good parallel tasks	1	1.75	2.5	3.25	4	4.75	5.5	6.25
Divisor for worse parallel tasks	1	1.5	2	2.5	3	3.5	4	4.5

2.2. Robustness

A resource allocation is robust if it meets a given performance criterion and is able to maintain this performance despite unexpected perturbations [3,23,27]. To quantitatively compare robustness among different possible resource allocations, three questions about robustness must be answered [2]: (1) *What behavior of the system makes it robust?* Our system is robust if all applications complete before a common deadline Δ. (2) *What uncertainties is the system robust against?* The uncertainty is the difference between the estimated execution time of each task and the actual data dependent execution time of each task. (3) *How is system robustness quantified?* In this study, we define robustness of a given resource allocation to be the smallest common percentage increase ($\boldsymbol{\rho}$) for all task execution times that causes the makespan to be equal to the deadline Δ. Thus, the goal is to maximize the robustness metric, i.e., maximize ρ. An example of a resource allocation with a robustness of 50% is shown in Fig. 3.

This is a different metric than makespan because we can have two solutions with an equal or similar makespan and two very different robustness metrics (as discussed Section 5.2). The same is true for any slack measure that is based on makespan. We want to optimize our robustness to be able to finish executing our application before the deadline despite possible variation in computation time.

2.3. Performance metric

In a real system, the execution times of all tasks will not be increased by the same percentage. The execution times of some tasks in a real system may even decrease. However, ρ can be used as a suitable measure for robustness in this environment—it can be viewed as a quality of service guarantee. The performance metric for this study is ρ. For a given, fixed, static resource allocation, the DAG makespan is monotonically non-decreasing as a function of ρ; i.e., as ρ increases, the execution times of the tasks increase, and, hence, the makespan of the DAG either increases or stays the same (but does not decrease).

Due to the complexity of the environment, it is difficult to identify a closed-form expression for ρ. Robustness is different than just increasing the makespan by ρ, because of the impact introduced by the inter-compute node data transfers. Thus, a binary search procedure is used to calculate ρ.

We calculate ρ based on a scale factor λ, where $\rho = \lambda - 1$. This factor is used to scale the execution times. If ρ is negative, then the resource allocation already exceeds the deadline, and hence, the resource allocation is not a valid solution.

We structure the binary search for λ as follows: to calculate an upper bound on λ, UB_λ, first, for each PE, sum the ETC values of the tasks assigned to that PE (this ignores communication times). Let π be the maximum value of these sums among all PEs. The starting value for the binary search is:

$$UB_\lambda = \Delta/\pi. \tag{1}$$

The binary search for the maximum λ will go between 0, and UB_λ. For each iteration of the binary search, we calculate the makespan (including communication). We continue until increasing all task execution times by λ, to the nearest hundredth gives a makespan of Δ.

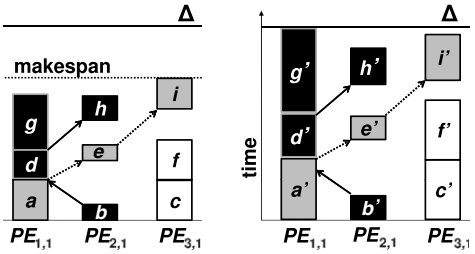


Fig. 3. An example of a resource allocation is shown. Each of the execution times for the tasks in (a) is increased by 50% with the results shown in (b). Note that the communication times do not increase, and the makespan for (b) (equal to Δ) is much less than the makespan in (a) increased by 50%. In (a), the makespan PE (the PE that determines the makespan) is $PE_{3,1}$. After all task execution times are increased by 50%, $PE_{3,1}$ is no longer the makespan PE. This example intuitively shows how the makespan is not a good measure of robustness.

(1) Generate a valid total ordering for all tasks (DAG_{list}).

(2) For 100 iterations:

(a) The first task in DAG_{list} is assigned to its estimated minimum completion time PE (Fig. 5).

(b) Communication is scheduled from the compute node (that has a copy of the data set) that can start the communication the earliest.

(c) The RAM of the compute node is allocated for this task's data sets (input and output).

(d) If it is possible then the task is parallelized across multiple PEs.

(e) The task used in step (a) is removed from DAG_{list}.

(f) The ready time of the PE on which the task is assigned is updated.

(g) Steps (a)-(f) are repeated until all the tasks have been mapped.

(g) Calculate robustness of complete resource allocation, and keep best solution across iterations.

(i) Mutate the total ordering using the procedure shown in Fig. 6.

(3) Output the best solution.

Fig. 4. Procedure for generating a resource allocation using Multicore MCT.

3. Heuristics

3.1. Common heuristic requirements

In this research, we require all the data to be in the RAM before a task can start executing. All the heuristics that are described in this section check that all the data is loaded into the RAM before a task can start execution.

When assigning a parallel task, our heuristics, will use all available PEs within a node. Initially, we experimented with all possible solutions (i.e., from one to the number of available PEs); however, we found out that heuristics would almost always use all the available PEs. Therefore, we limited the possibilities of one PE or all idle PEs to reduce the run time of the heuristic.

3.2. Multicore minimum completion time

The *M*ulticore *M*inimum *C*ompletion *T*ime (*MC MCT*) heuristic estimates the completion time of each resource allocation, and assigns each task to the machine that gives it its minimum completion time. The MC MCT uses a known total ordering (linear ordering of all tasks in DAG that obey the precedence constraints) to generate a resource allocation.

This heuristic is based on the concept used in the Minimum Completion Time [14]. The main difference in this heuristic is that we take into account the inclusion of RAM, the parallelization of tasks, and we use an already generated valid total ordering to assign the tasks.

The procedure for MC MCT is shown in Fig. 4. The procedure used to calculate the estimated completion time is shown in Fig. 5.

(1) Let $transfer_{time} = 0$.

(2) For a given task t_i on PE j (PE_j)

(a) For each data set dat (satellite and inter-task) required by t_i at PE_j:

(i) Determine the smallest transfer time from all possible sources to move dat to the RAM of the compute node associated to PE_j.

(ii) Add the transfer time from a(i) to $transfer_{time}$ (ignoring availability of communication channels of compute nodes).

(3) For each parent of task t_i, determine the maximum completion among all the parent tasks ($\boldsymbol{max_{source}}$).

(4) Estimated completion time of t_i on PE_j is equal to the sum of $transfer_{time}$, the maximum of max_{source} and the ready time of PE_j for t_i, and $ETC(i, j)$.

Fig. 5. Procedure for estimating a completion time in our environment.

(1) A random task $\boldsymbol{t_{rand}}$ is selected for mutation.

(2) A valid range where t_{rand} can be moved in the ordering without violating any precedence constraint is determined (i.e., after all predecessor tasks and before all successor tasks) [20].

(3) A random position within this range is selected, and t_{rand} is moved to this new position.

Fig. 6. Procedure to modify a total ordering used in MC MCT and Genitor.

The procedure for modifying a total ordering (based on the mutation in [30]) used to create multiple MC MCT solutions is shown in Fig. 6. We monitored the generated solutions and in our trials we did not have any repeated solutions. However, we did not limit the mutation procedure to generate unique total ordering. This heuristic was used mostly as a baseline heuristic for comparison. Therefore, we do not think avoiding duplicate solutions would greatly improve the performance of the heuristic.

3.3. Multicore Random Resource Allocation

The Multicore Random Resource Allocation heuristic uses an arbitrary total ordering and assigns the tasks (in order) to a randomly selected PE. The procedure is the same as that of MC MCT except that in step 2(a) of Fig. 4 we assign tasks to PEs randomly.

3.4. Heterogeneous Robust Duplication

Overview

*H*eterogeneous *R*obust *D*uplication (*HRD*) is based on the concept in the Highest Critical Parents with Fast Duplication heuristic introduced in [13]. The algorithm has a listing phase, where tasks have their priority computed and are inserted into a queue based on that priority. The algorithm then has a scheduling phase, where the tasks are assigned to PEs in order of the queue. The significant differences between this adaptation of the algorithm and the version in [13] are (a) optimizing ρ therefore the equations for the scheduling phase and the mapping phase have to be adapted for our robustness metric, (b) we have to take into consideration the memory management problem of moving data from an HD to RAM, and (c) we need to consider parallelizable tasks.

The listing phase

Let $TT(dat_{i,j}, memory_\alpha, memory_\beta)$ be the *t*ransfer *t*ime of $dat_{i,j}$ between the memory (HD or RAM) of the source compute node α and the memory (HD or RAM) of the destination node β. We can calculate the average transfer time ($TT_{average}(dat_{i,j}, \alpha, \beta)$) between any pair of compute nodes as

$$\begin{aligned} TT_{average}(dat_{i,j}, \alpha, \beta) = {} & (TT(dat_{i,j}, RAM_\alpha, HD_\beta) \\ & + TT(dat_{i,j}, HD_\alpha, RAM_\beta) \\ & + TT(dat_{i,j}, HD\alpha, HD_\beta) \\ & + TT(dat_{i,j}, RAM_\alpha, RAM_\beta))/4. \end{aligned} \quad (2)$$

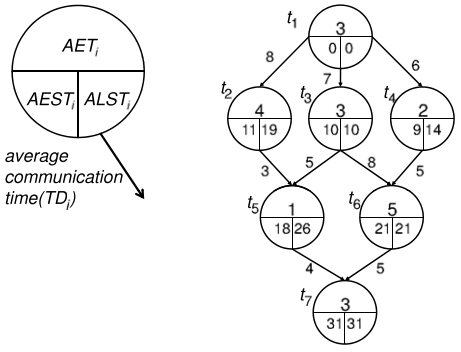


Fig. 7. An example of the AEST and ALST computation. AEST is calculated first starting from t_1 to t_7. After all AESTs are calculated, ALST of t_7 is set to AEST of t_7 (in this case 31), and ALST of all the tasks from t_7 to t_1 is calculated.

(1) Set list L and stack S to null.
(2) For each application DAG
  (a) Traverse DAG downward (starting at entry task $\boldsymbol{t_{entry}}$), computing AEST for each task
  (b) Traverse DAG upward (starting at exit task $\boldsymbol{t_{exit}}$), computing ALST for each task.
  (c) Identify all critical tasks (where $ALST = AEST$).
  (d) Push the critical tasks on the stack ($\boldsymbol{S}$) in descending order of their ALST.
(3) Sort S in descending order of their ALST
(4) While S is not empty do
  (a) If the task at the top of the stack S has a parent that is not in L then push the parent on S (so that the parent is at the top of the stack).
  (b) Else pop S and add at the end of L.

Fig. 8. Procedure used to generate the HRD list.

The *a*verage *c*ommunication *t*ime for transferring a data item ($\boldsymbol{dat_{i,j}}$) created by t_i and consumed by t_j between any pair of compute nodes ($\boldsymbol{ACT}(\boldsymbol{dat_{i,j}})$) is calculated as

$$ACT(dat_{i,j}) = \sum_{\alpha=1}^{N}\sum_{\beta=1}^{N}\left[TT_{average}(dat_{i,j}, \alpha, \beta)\right]/N^2. \quad (3)$$

The HRD calculates two values for each task, the *A*verage *E*arliest *S*tart *T*ime (*AEST*) and the *A*verage *L*atest *S*tart *T*ime (*ALST*). We traverse down the DAG computing $AEST(t_i)$ for each task t_i. We define the *a*verage *e*xecution *t*ime of t_i as $\boldsymbol{AET}(\boldsymbol{t_i})$:

$$AET(t_i) = \sum_{j=1}^{N} ETC(i,j)\cdot v_j/M. \quad (4)$$

Let $\mathbf{pred}(t_i)$ be the set of predecessor tasks for t_i in the DAG, and $\mathbf{succ}(t_i)$ be the set of successor tasks. We can calculate $AEST(t_i)$ recursively:

$$AEST(t_{entry}) = 0, \quad (5)$$

$$AEST(t_i) = \max_{t_j \in pred(t_i)}[AEST(t_j) + AET(t_j) + ACT(dat_{i,j})]. \quad (6)$$

To compute $ALST(t_i)$ recursively, we traverse up the DAG, for each task t_i:

$$ALST(t_{exit}) = AEST(t_{exit}), \quad (7)$$

$$ALST(t_i) = \min_{t_j \in succ(t_i)}[ALST(t_j) - ACT(dat_{i,j})] - AET(t_i). \quad (8)$$

Tasks along the critical path (*critical tasks*) have ALST equal to AEST. An example of this is shown in Fig. 7.

In the next part of the listing phase, a list $\boldsymbol{L}$ is built for the mapping phase, using the procedure from [13] shown in Fig. 8. The goal of this total ordered list L (which respects precedence constraints) procedure is to consider the critical task path (beginning with t_{entry} and ending with t_{exit}) as soon as possible.

(1) Generate the HRD list using the procedure in Fig. 8.
(2) For each task in the HRD list
  (a) A task (t_{map}) is de-queued from L
  (b) For t_{map} on each PE_j
    (i) Determine the best allocation based on Fig. 10.
    (ii) Communication is scheduled from the compute node (that has a copy of the data set) that can start the communication the earliest.
    (iii) The RAM of the compute node is allocated for this task's data sets (input and output) as soon as possible.
    (iv) If it is possible then the task is parallelized across multiple PEs.
    (v) The ready time of the PE on which the task is assigned is updated.
(3) Output the solution.

Fig. 9. Procedure used to map tasks to machines for the HRD heuristic.

(i) Calculate the robustness (considering communication, memory allocation, and parallelization) of assigning t_{map} to PE_j.
(ii) Calculate the completion time (considering communication, memory allocation, and parallelization).
(iii) We compare this allocation of t_{map} to the best allocation of t_{map}.
(iv) If the DTS is large enough to hold the execution time of t_{map}, then attempt duplication. Keep the duplicated critical task if the robustness is improved.

Fig. 10. Procedure used to assign a task to a machine in the HRD heuristic.

Mapping phase

For the mapping phase, a task $\boldsymbol{t_{map}}$ is dequeued from L. Assign t_{map} to the PE that results in the maximum robustness value. Multiple PEs may have the same robustness value. If ties occur then we select the PE with the minimum completion time, equal to the estimated ready time of the PE plus the execution time of t_{map} on that PE. We repeat this until the queue is empty.

Reducing the communication may help increase ρ because this reduction allows t_{map} and subsequent tasks to start and complete sooner. It also reduces the network congestion, which can also lead to an earlier completion time. In this study, we consider duplicating a parent task on the same compute node as the child. For this study, we consider one possible duplication scheme. In this duplication scheme, we are not doing an exhaustive search of possible duplication spots.

If there is an idle time slot just prior to the execution of a task on a given PE it is denoted as the *d*uplicate *t*ime *s*lot (*DTS*). The DTS for each PE is the start time of t_{map} minus the time when the PE finishes executing the task it has before t_{map} starts execution. If the DTS is large enough to hold the execution of the critical parent then duplication is attempted, thus, eliminating the need to incur the communication cost of the critical parent's data set. For this study, we attempted the duplication if the DTS was large enough. When we evaluate the robustness of a given resource allocation with a duplicated task, the duplicated task is invalidated if all of its data is not available (or cannot be made available) at its scheduled start time or it does not improve robustness. The procedure for the HRD is shown in Fig. 9 (see Fig. 10).

3.5. Dynamic available tasks critical path

Base version

The *D*ynamic *A*vailable *T*asks *C*ritical *P*ath (*DATCP*) heuristic is based on the concept used in Dynamic Critical Path (DCP) heuristics [18]. It computes the critical path for every task and chooses the ordering for mapping tasks using this critical path value. The significant differences between this adaptation of the algorithm and the version in [18] are (a) optimizing ρ therefore the equations for the scheduling phase and the mapping phase have to be adapted for our robustness metric, (b) we have to take into consideration the memory management problem of moving data from an HD to RAM, and (c) we need to consider parallelizable tasks. Without loss of generality, assume that all of the entry nodes of all application DAGs have a common predecessor that is the pseudo-node entry

(1) Let $t_i = t_{pseudo\ exit}$
(2) Calculate average task execution time ($AET(t_i)$).
(3) For each SD_x needed by t_i, calculate the estimated transfer time (using $ACT(SD_x)$ Equation 3).
(4) For each TD_j needed by t_i, calculate the estimated transfer time to successor nodes (using $ACT(TD_j)$ Equation 3).
(5) Determine the maximum time from any successor (child) node to the $t_{pseudo\ exit}$ ($\boldsymbol{max_{time}}$).
(6) Critical path value is sum of each individual TD and SD transfer time, max_{time}, and $AET(t_i)$.
(7) Select another task (whose successors have a calculated critical path value) and go to step (2) until all tasks are processed.

Fig. 11. Procedure used to calculate the critical path of the DATCP heuristic.

(1) Calculate the critical path for each application (using the procedure shown in Fig. 11).
(2) Dynamically create a list of all tasks available for mapping (i.e., predecessors have been mapped).
(3) Determine the task with the longest critical path from the list of available tasks.
(4) For task t_map determined in (3),
  (a) Determine the allocation using the procedure in Fig. 13.
  (b) Communication is scheduled from the compute node (that has a copy of the data set) that can start the communication the earliest.
  (c) The RAM of $node_{max}$ is allocated for this task's data sets (input and ouput) as soon as possible.
  (d) If it is possible then the task is parallelized across multiple PEs (see Section 2.1).
  (e) The ready time of the PE(s) on which the task is assigned is updated.
  (f) Remove task t_i from list.
(5) Repeat steps (2)–(4) until all tasks are mapped.

Fig. 12. DATCP heuristic procedure used to generate a resource allocation.

(i) For each PE j (each j is unique across compute nodes), calculate the robustness of assigning t_{map} to PE_j (including communication, memory, and task parallelization).
(ii) Find the compute node j with the highest number of maximum-robustness-value PEs (it is possible for other PEs to have the same robustness value). This compute node is denoted $\boldsymbol{node_{max}}$.
(iii) Map task to any one of the PEs with the maximum robustness in $node_{max}$.

Fig. 13. Procedure used to map tasks to machines for the DATCP heuristic.

task $\boldsymbol{t_{pseudo\ entry}}$. Similarly, assume that there is a unique exit node $\boldsymbol{t_{pseudo\ exit}}$. The calculation of the critical path is a recursive process that begins with $t_{pseudo\ exit}$ and finishes at $t_{pseudo\ entry}$. Each task calculates its estimate of the average time to reach $t_{pseudo\ exit}$ and then passes this time value to its predecessor tasks. The execution time for a task is calculated using the average values across all compute nodes (calculated using Eq. (4)).

The pseudo-code for the critical path calculation method is in Fig. 11. After calculating the critical path, the next step of the heuristic determines the list of available tasks, which are entry tasks or tasks whose predecessors have been mapped. The available task with the longest critical path time is then mapped to the PE that maximizes the robustness value. That task is then removed from the list of available tasks and the process is repeated until all tasks are mapped. The pseudo-code for this algorithm is listed in Fig. 12.

Variation

A variation of this heuristic (*DATCP-HRD*) was implemented to determine the effect of breaking ties between two allocations with the same robustness. DATCP assigns the tasks to a PE on the compute node with the maximum number of PEs with the same robustness, and HRD assigns it to the PE that has the minimum completion time. The DATCP-HRD variation is similar to the DATCP heuristic except that step 4(a) uses Fig. 9 instead of 13.

3.6. Multicore dynamic levels

The *M*ulticore *D*ynamic *L*evels *MC DL* heuristic is based on the concept in [25]. The significant differences between this adaptation of the algorithm and the version in [25] are (a) we have to take into consideration the memory management problem of moving data from an HD to RAM, and (b) we need to consider parallelizable tasks.

(1) Calculate the static levels for all tasks (Equation 10).
(2) While tasks remain unassigned
  (a) For each task-PE pair calculate the dynamic level (Equation 11).
  (b) For task-PE pair with the highest dynamic level, assign this task to the corresponding PE.
  (c) Communication is scheduled from the compute node (that has a copy of the data set) that can start the communication the earliest.
  (d) The RAM of the compute node is allocated for this task's data sets (input and ouput) as soon as possible.
  (e) If it is possible then the task is parallelized across multiple PEs.
  (f) The ready time of the PE(s) on which the task is assigned is updated.
(3) Output the solution.

Fig. 14. Procedure used to assign tasks to machines using dynamic levels.

In the MC DL heuristic, the *static level* of a task is computed as an approximate time from the task node to the exit node along the worst-case path based on average task execution times. Each task has its own static time that represents how much computation is needed to reach the exit task. This static time does not take into account any allocation and does not change once it is calculated. A *data arrival time* for a task is defined as the time when all required input data for that task have arrived at a destination node and is based on some given partial allocation beginning at the entry node. The data arrival time tells us when the task will have the data that it needs to begin executing. The *dynamic level* is the static level minus the maximum of the data arrival time and the ready time of a PE (the time when it finishes executing its previous task), all based on the partial allocation. The mappable-task/machine pair with the highest dynamic level value is selected for mapping. Intuitively, the dynamic level value should give more priority to those tasks that have a large amount of computation before they are done and to the tasks that can start early. This dynamic level depends on the state of the system due to previous task-to-PE assignments. The difference between our work and [25] is that this version assigns memory, schedules data transfers, and employs task level parallelism in a multicore environment.

The static level $SL(t_i)$ is calculated recursively using the following equation:

$$SL(t_{exit}) = AET(t_{exit}) \tag{9}$$

$$SL(t_i) = \max_{t_j \in succ(t_i)} \left[SL(t_j)\right] + AET(t_i). \tag{10}$$

For each task (t_i) on a PE (PE_j), the current dynamic level is calculated before every task is assigned. Let $\boldsymbol{S}$ be the state of processing and communication resources, $\boldsymbol{DA(t_i, PE_j, S)}$ be the earliest time that all the data transfer to t_i (on PE_j) from its predecessors are guaranteed to be completed, and $\boldsymbol{TF(PE_j, S)}$ be the time when the task assigned to PE_j finishes. The dynamic level of t_i on PE_j at state S ($\boldsymbol{DL(t_i, PE_j, S)}$) is calculated as follows:

$$DL(t_i, PE_j, S) = SL(t_i) - \max[TF(PE_j, S), DA(t_i, PE_j, S)]. \tag{11}$$

The procedure for the Multicore Dynamic Levels is shown in Fig. 14. The MC DL heuristic performs well for certain scenarios despite the fact that it does not explicitly consider robustness.

3.7. Multicore Genitor

The Multicore Genitor heuristic is based on the concept of the Genitor heuristic [31]. The version we adapted of Genitor is based on the GA concepts shown in [24] and Genitor concepts in [31]. The significant differences between this adaptation of the algorithm and the version in [31] are (a) optimizing ρ therefore the

equations for the scheduling phase and the mapping phase have to be adapted for our robustness metric, (b) we have to take into consideration the memory management problem of moving data from an HD to RAM, and (c) we need to consider parallelizable tasks.

The Genitor is a steady state heuristic that only does one crossover and mutation operation per iteration and uses an ordered population to keep the best chromosomes (encountered during the simulation) in the population. In this study, we used the robustness as the fitness function for the chromosomes. The results of the crossover and mutation are evaluated and inserted in the population based on their robustness.

The chromosome used in the Multicore Genitor is based on the work in [30]. The chromosome in [30] has two strings: a mapping string and a scheduling string. The mapping string is a vector of length $LN = \sum_{\forall k} T_k$ (recall that T_k is the number of tasks in application k), where the ith entry in the chromosome represents task t_i (assume that LN tasks are uniquely numbered from 1 to LN), and the value of the entry represents the PE where t_i is assigned. The scheduling string is also of length LN; however, the jth entry represents the jth task in a particular total ordering. The difference between the Genitor heuristic in [30] and this paper is that Genitor in this paper maximizes robustness, stages data sets to RAM, parallelizes tasks, and determines satellite placement. The full procedure for the Multicore Genitor is shown in Fig. 15, and the procedure for crossover is shown in Fig. 16.

1) An initial population of chromosomes (number is determined empirically) is generated and evaluated.
   (i) Seeds of Multicore Dynamic Levels heuristic and DATCP are generated.
   (ii) The remaining chromosomes are created by using the total ordering generated by the Multicore Dynamic Levels heuristic as an initial ordering for the MC MCT heuristic (shown in Fig. 4).
2) While there are less than 1000 iterations without improvement, repeat the following procedure.
   (i) A pair of parents are selected for crossover and mutation using linear bias (1.5 determined experimentally) [24].
   (ii) Two offspring are generated using one-point crossover (for both the mapping and scheduling string). For the scheduling string, the crossover procedure is shown in Fig. 16.
   (iii) For each offspring, there is a 1% probability of mutating each field (determined empirically) in the chromosome. For the scheduling string, the procedure for mutation is shown in Fig. 6.
   (iv) The offspring are evaluated and ranked into the population displacing the worst chromosome.
3) The output is the best solution.

Fig. 15. Procedure used for resource allocation by the Multicore Genitor.

1) A random position in the scheduling string is selected for a one-point crossover.
2) For parent A, create ordered list $\boldsymbol{list_A}$ with tasks from bottom part of its scheduling string.
3) For parent B, create ordered list $\boldsymbol{list_B}$ with tasks from bottom part of its scheduling string.
4) We create a copy of parent A (denoted **offspring A**), and parent B (denoted **offspring B**).
5) For offspring A, we change its scheduling string so that the tasks in $list_B$ follow the same relative ordering in $list_B$ and offspring A.
6) For offspring B, we change its scheduling string so that the tasks in $list_A$ follow the same relative ordering in $list_A$ and offspring B.

Fig. 16. Procedure used for crossover in the Multicore Genitor heuristic.

4. Related work

The research that discusses scheduling DAGs on multiprocessor systems is extensive, e.g., [17–19,25,28,29,32,20]. In this section, a comparison between our heuristics and existing heuristics is presented.

We used the methodology described in [3] to derive a robustness metric that quantifies how robust our resource allocation is to changes in task execution time. The robustness metric proposed in this study is very different from the example used in [3].

The authors in [7] propose a system that uses random variables to represent both computation and communication. Robustness in the paper is increased by decreasing the average makespan and decreasing the standard deviation of the makespan. In our study, we use estimated execution times. We also do not have any additional information about the probability density function of our task execution times so we use a percentage.

The Modified Critical Path (*MCP*) algorithm developed in [32] was designed for homogeneous systems and considers only one application DAG. The heuristic determines the latest possible start time of each task (constrained by the critical path length) and then creates a list of tasks in the increasing order of these times. Tasks are selected for mapping in the order of the list. The selected task is then mapped to the machine that allows the earliest start time. Thus, the heuristic attempts to start critical path tasks as early as feasible. There are several differences between [32] and this study. We focus on robustness against uncertainty in execution times and consider a heterogeneous computing environment.

The authors in [25] developed the Dynamic Level Scheduling (*DLS*) heuristic. The static level of a task is computed as an approximate time from the task node to the exit node along the worst-case path on a heterogeneous system. A data arrival time for a task is defined as the time when all required input data arrive for a task at a destination node. The dynamic level is the static level minus the maximum of data arrival time and the time when the processing element is ready to execute a new task. The mappable task/machine pair with the highest dynamic level value is selected for mapping. While our heuristic builds on this work its performance metric is different. The performance metric in [25] was makespan, and the paper did not consider uncertainties or robustness. As indicated earlier, minimizing makespan is not the same as maximizing our robustness measure for DAGs.

The authors in [20] present the Recursive Convex Cluster Algorithm for resource allocation in a parallel and distributed system. We focus on uncertainty in computation while [20] focuses on uncertainty in communication and because of this their techniques are not appropriate for our problem.

The Dynamic Critical Path (*DCP*) heuristic developed in [18] calculates the *Ab*solute *E*arliest *S*tart *T*imes (*AbEST*) and *Ab*solute *L*atest *S*tart *T*ime (*AbLST*) for each task. A task is defined to be on the critical path if its AbEST equals its AbLST. At each mapping event, mappable tasks update their AbEST and AbLST to determine which task is on the critical path. The critical path task is then mapped to a compute node that minimizes the *E*arliest *S*tart *T*ime (*EST*) of the task and the EST of its successor tasks. The DATCP heuristic differs from the DCP heuristic in that the DCP heuristic does not have to deal with uncertainty in task execution times and its performance goal is makespan rather than robustness.

The work in [24] considers a heterogeneous *ad hoc* grid used to compute an application composed of communicating sub-tasks. Both this and our study consider the mapping problem of DAGs in a heterogeneous computing environment; the heuristics in [24] minimize the average battery power consumed while meeting a makespan constraint. This paper focuses on maximizing the ability of a resource allocation to tolerate the uncertainty of execution times, while the work in [24] does not consider uncertainty. The minimization of battery power consumed, as studied in [24], is different than maximizing robustness presented in this study.

5. Results

5.1. Simulation setup

Each simulation run consists of 50 trials each with 50 unique applications, and each application is composed of 8–16 tasks where the number of tasks is chosen with uniform probability. We generate DAGs with unique entry and unique exit tasks. The DAGs have a maximum fanout of 3, and a maximum fanin of 3, similar to the example in Fig. 7. This type of DAG was used because it is similar to the DAGs encountered in real-world environments like DigitalGlobe.

The size of the SD and TD sets used in this simulation varies from 1 to 20 GBytes. All compute nodes have 160 GBytes of RAM (152 are used for staging data and the remaining 8 are assumed to be used to buffer data in and out of the local HD). We assume that each network link has a bandwidth of 512 Mbytes per second.

The HC system is composed of four compute nodes. Compute node 1 has eight PEs, compute node 2 has four PEs, and finally compute nodes 3 and 4 have two PEs. Note that all compute nodes are sorted in the descending order based on the number of PEs. If k is greater than j, then the ETC of computing t_i on compute node j (PE running individually) is less than or equal to its ETC on compute node k. Note that even if two nodes have the same number of PEs, they may still have different ETC values. The intuition behind ordering the ETC-per-GByte in this manner is that newer systems will have more cores per multiprocessor, and a PE in this multiprocessor will be faster than a PE in an older multiprocessor.

The ETC values of a task are calculated by adding all the incoming data sets (SD and TD in GBytes) and multiplying this sum by an ETC-per-GByte value generated using the coefficient of variation based method described in [4]. A high-task/high-compute-node heterogeneity [4] is used to determine the ETC per GByte value for the tasks within the DAGs. The heterogeneity values used to generate the ETCs per GByte are $V_{task} = \frac{\sigma_{task}}{\mu_{task}} = 0.4$ and $V_{machine} = \frac{\sigma_{machine}}{\mu_{machine}} = 0.3$, and the mean ($\mu_{task}$) is 1 s/Gbyte. Half of the tasks in the applications are parallelizable, with half of those having "good" parallelism and half having "worse" parallelism (see Table 1 in Section 2).

Additional parameters were created to vary the robustness: the mean task computation time and task communication bandwidth. The first parameter is $\alpha_{comp} \in \Re$, used to scale the computation by a factor of α_{comp}; and the second parameter is $\alpha_{comm} \in \Re$, used to scale the communication bandwidth by a factor of $1/\alpha_{comm}$. Note that an increase in the α_{comm} value will lead to a higher bandwidth and therefore a smaller communication time. The tight and loose deadlines are used to evaluate the heuristics in a situation where it is easy to meet the deadline and a situation where it is not easy. They were determined experimentally for each simulation scenario. The six different scenarios simulated in this study are shown in Table 2, where each scenario involved 50 tasks. The error bars in each results figure show the 95% confidence intervals.

5.2. Simulation results

Medium computation and high communication

In this subsection, we consider two scenarios with medium computation and high communication (i.e., $\alpha_{comp} = \alpha_{comm} = 1$). Thus, in these scenarios, the communication times are long compared to the computation times.

In Fig. 17, we can observe that both DATCP variations have the best performance in terms of makespan and robustness. It is important to remember that the fitness functions used to assign tasks to PEs in DATCP and DATCP-HRD are very different; however, their performance is comparable. The scenario with α_{comp} and $\alpha_{comm} = 1$ has long communication times compared to computation times.

Table 2
Simulation scenarios and their respective configurations.

Scenario	α_{comp}	α_{comm}	Δ
1	Medium (1)	High (1)	Loose (15,500)
2	Medium (1)	High (1)	Tight (10,500)
3	Medium (1)	Medium (1/3)	Loose (8000)
4	Medium (1)	Medium (1/3)	Tight (6000)
5	High (2)	Low (1/6)	Loose (8000)
6	High (2)	Low (1/6)	Tight (6000)

The results with $\alpha_{comp} = 1$, $\alpha_{comm} = 1$, and $\Delta = 10,500$ (tight deadline) show the DATCP and DATCP-HRD heuristics significantly outperforming the other heuristics (both in makespan and robustness). In this scenario, the HRD was barely able to keep a robustness performance of 0 (makespan equal to deadline), the MC MCT was on average not able to meet the deadline constraint, and the MC Random heuristic did not make the deadline on any run.

For both of the scenarios, the DATCP heuristics are able to outperform the other heuristics because they use more information than the other heuristics when determining the total ordering. In our study, the DATCP provides a better total ordering because (a) it does not use estimates, (b) it uses the state of the HC system when it determines the next task that needs to be assigned, (c) uses critical path to determine the ordering, and (d) uses robustness to allocate tasks to PEs. The MC DL also uses the updated information about the system state when it determines the ordering; however, it does not use robustness to guide the resource allocation. The MC Random heuristic had the worst performance and the highest makespan, because it does not consider communication or computation. Therefore, it will often assign tasks from one application across multiple compute nodes, and these tasks may be unable to start executing quickly because of large communication delays in transferring required data. The performance of Genitor is as good as or better than the other heuristics because it contains the resource allocations generated by the DATCP and MC DL heuristics in the initial population.

Medium computation and medium communication

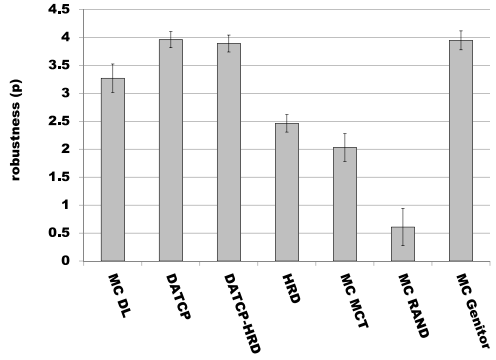
For the results in Fig. 18, we reduced the average communication times by a factor of 1/3 (i.e., $\alpha_{comp} = 1$ and $\alpha_{comm} = 1/3$). This causes the scheduling gaps created by cross-compute-node communication to be reduced significantly.

In Fig. 18 (loose deadline), the MC Genitor, DATCP and DATCP-HRD outperform the other heuristics with regard to makespan; however, in terms of robustness the MC Genitor, DATCP and DATCP-HRD have comparable performance to MC DL heuristics. The results in Fig. 18(b) show that even with the reduced communication times, when the deadline is tightened, the DATCP and DATCP-HRD outperform the other heuristics (both in makespan and robustness). In this scenario, the MC Random heuristic was the worst performing heuristic; however the performance in the medium/medium scenario vs. medium/high scenario improved significantly. This was due to a reduction in the time it takes to move a data set. It is interesting to see in this example that the makespan of the Genitor heuristics is on average larger than the average of the DATCP heuristic. This indicates that the MC Genitor heuristic increased the makespan when it tries to maximize the robustness.

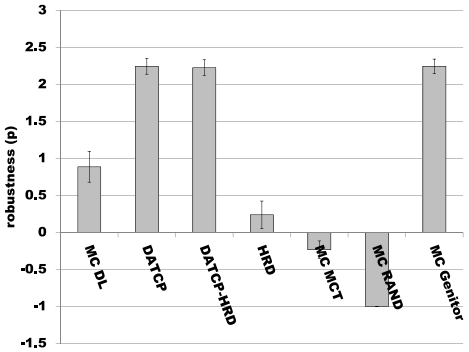
High computation and low communication:

For the results in Fig. 19, we increased the computation time by a factor of two and reduced the average communication times by a factor of six (i.e., $\alpha_{comp} = 2$ and $\alpha_{comm} = 1/6$). This drastically reduces the gaps created by cross-compute-node communication. This causes the makespan and robustness to be even more closely correlated, as discussed later.

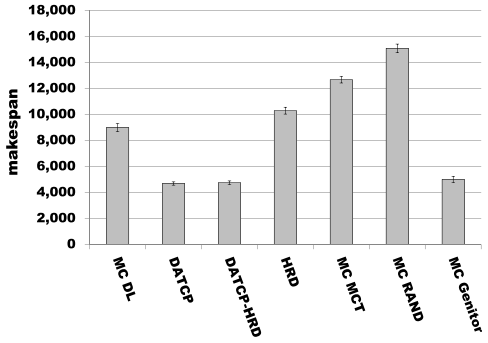
For the loose deadline scenario in Fig. 19, the MC DL heuristic and the MC MCT heuristics have a comparable performance to



(a) Robustness.

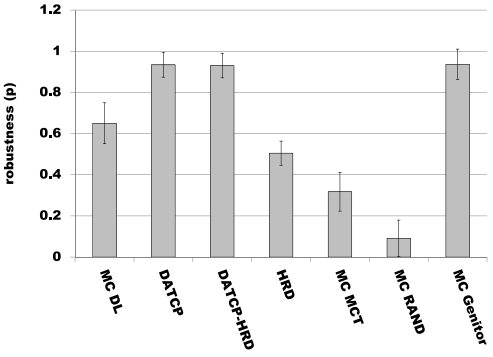


(b) Robustness.

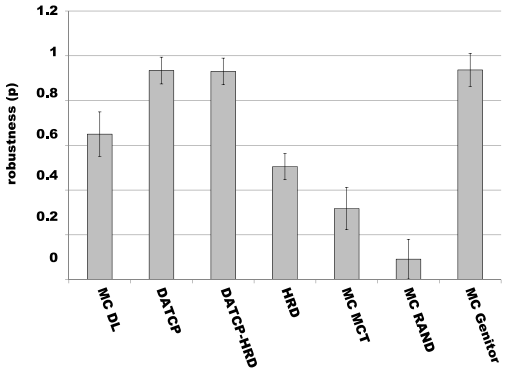


(c) Makespan.

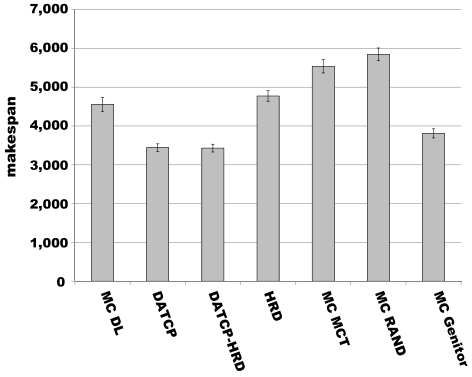
Fig. 17. Medium computation and high communication: robustness results for the heuristics defined in Section 3 with $\alpha_{comp} = \alpha_{comm} = 1$ and (a) $\Delta = 15,500$ (loose) and (b) $\Delta = 10,500$ (tight). Makespan for case (a) shown in (c). Makespan in case (b) has similar relative performance.



(a) Robustness.

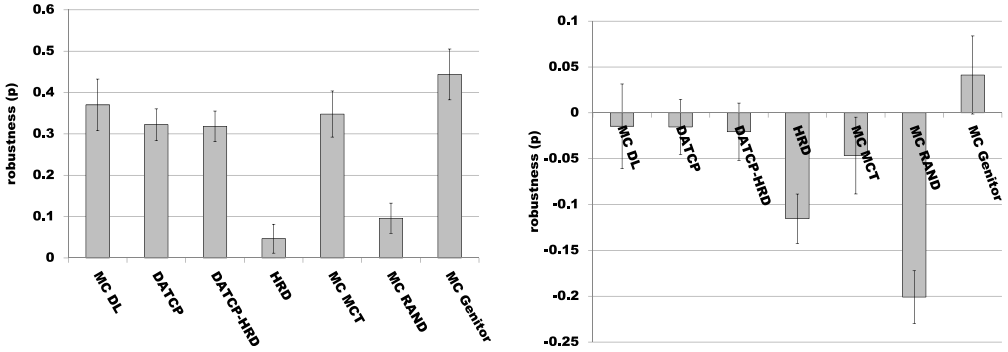


(b) Robustness.



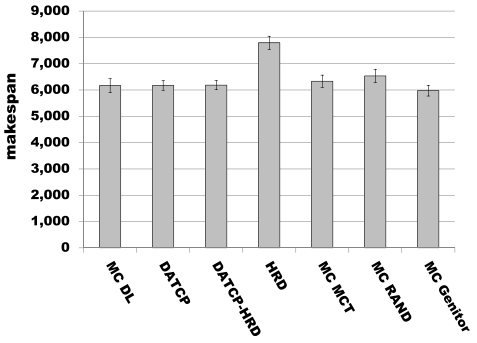
(c) Makespan.

Fig. 18. Medium computation and medium communication: robustness results for the heuristics defined in Section 3 with $\alpha_{comp} = 1$ and $\alpha_{comm} = 1/3$ and (a) $\Delta = 8000$ (loose) and (b) $\Delta = 6000$ (tight). Makespan for case (a) shown in (c). Makespan in case (b) has similar relative performance.



(a) Robustness.

(b) Robustness.



(c) Makespan.

Fig. 19. High computation and low communication: robustness results for the heuristics defined in Section 3 with $\alpha_{comp} = 2$ and $\alpha_{comm} = 1/6$ and (a) $\Delta = 8000$ (loose) and (b) $\Delta = 6000$ (tight). Makespan for case (a) shown in (c). Makespan in case (b) has similar relative performance.

the more elaborate heuristics. In fact, MC DL has overlapping confidence intervals for robustness with the MC MCT, DATCP, and DATCP-HRD heuristics. In this scenario, the HRD and MC random are the worst performing heuristics. At the high/low scenario, MC MCT is better because communication is very small therefore it only relies on computation. Additionally, MC MCT evaluates 50 different total orderings and picks the best among those.

In Fig. 19(b), we see that the performance of DATCP is comparable to the best performing heuristics when we make the deadline tighter. In this case, the performance of MC DL, MC MCT, DATCP, and DATCP-HRD is comparable in robustness (overlapping confidence intervals). This occurs because (1) the high ratio of computation to communication causes the increase in computation to be a larger factor than communication, and (2) the deadline is very tight. When we compare the averages for the scenario shown in Fig. 19(b) we can see that one heuristic had a positive average, i.e., MC Genitor heuristic.

Comparison of makespan vs. robustness with loose deadlines

In Fig. 20, the MC MCT heuristic was used to generate 20,000 resource allocations for a single trial, for a given scenario. The deadlines were made loose to avoid the heuristic from only finding solutions that had a makespan that was close to the deadline. The α_{comp} and α_{comm} values for these scenarios were selected in relation to the average computation time (average ETC per GB multiplied by the average size) and the average communication time of a task over the network (average size divided by network bandwidth). For example, when $\alpha_{comp} = 2$ and $\alpha_{comm} = 1/6$. This means that our ratio of computation to communication changes from an initial $x : y$ to $2x : y/6$. The three cases shown represent a scenario where communication is dominant 20(a), communication and computation are comparable 20(b), and computation is dominant 20(c).

It is interesting to see that as the computation time to communication time ratio increased, the robustness as a function of makespan approaches the relationship of $(\Delta/makespan) - 1$. In Fig. 20, a line showing $(\Delta/makespan) - 1$ is graphed to show how makespan and robustness are related based on the computation time and communication time ratio across the three cases (note $(\Delta/makespan) - 1$ is the same for all cases). This line would be equal to ρ if there was no communication and interdependency between tasks (this is approximated in Fig. 20(c)). In general, we can have two solutions with an equal or similar makespan and two very different robustness metrics.

However, when communication is predominant then there is a large deviation between $(\Delta/makespan) - 1$ and heuristic robustness. The results in Fig. 20 indicate that using our robustness metric is better than makespan in systems where communication is greater or comparable to computation. This is because of the better performance of robustness based heuristics over heuristics that just use makespan when there is more communication than computation. When communication times are small or close to negligible compared to computation times the makespan and the robustness metric are correlated.

Summary

As shown in Fig. 20(a), when computation is medium ($\alpha_{comp} = 1$) and communication is high ($\alpha_{comm} = 1$) the robustness and makespan are very loosely correlated. Therefore, using the robustness metric to guide a heuristic in this scenario is important. As shown in Fig. 20(c), when computation is high ($\alpha_{comp} = 2$) and communication is low ($\alpha_{comm} = 1/6$) the makespan and robustness are highly correlated. Therefore, in general, using the robustness metric does not yield a significant improvement over just using makespan. The MC Genitor heuristic performed well in all environments; however, this is expected because it uses the DATCP, DATCP-HRD, and MC-DL heuristics as seeds.

5.3. Heuristic execution time

The heuristics that ignored the robustness (MC MCT, MC RAND, and MC-DL) were able to generate a resource allocation significantly faster those that considered robustness. The MC MCT, MC

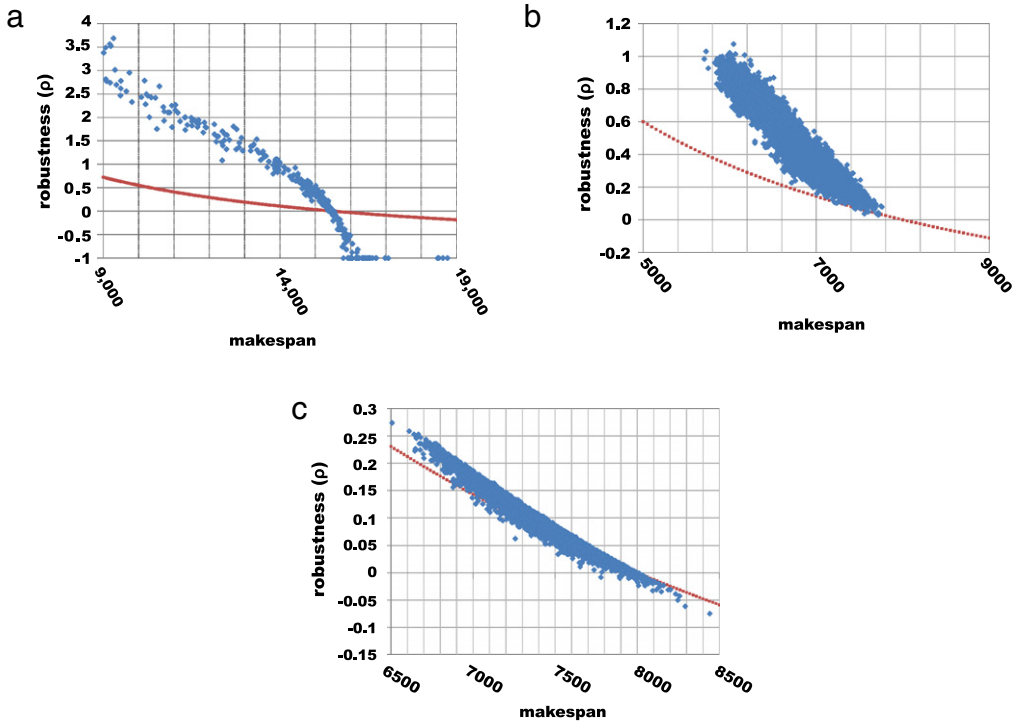


Fig. 20. Scatter plot of makespan vs. robustness for 20,000 resource allocations of the MC MCT heuristic in three different scenarios: (a) $\alpha_{comp} = 1$, $\alpha_{comm} = 1$, and $\Delta = 15,500$; (b) $\alpha_{comp} = 1$, $\alpha_{comm} = 1/3$, and $\Delta = 8000$; and (c) $\alpha_{comp} = 2$, $\alpha_{comm} = 1/6$, and $\Delta = 8000$. Additionally, a line showing $(\Delta/makespan) - 1$ (square markers) was graphed in (a)–(c) to show how makespan and robustness are related based on the computation time and communication time ratio across the three cases (note $(\Delta/makespan) - 1$ is the same for all cases). Note that the x and y axes change for each graph.

RAND, and MC-DL all took approximately two minutes to run. Heuristics that used robustness to calculate the schedule ran significantly slower; DATCP took on average approximately twelve minutes, HRD took on average ten minutes, DATCP-HRD took on average about 20 min, and Genitor took 60 min. Because these heuristics would be executed offline in this static environment, these longer runtimes are acceptable.

6. Conclusions

This study focused on using heuristics to allocate resources for multiple applications (formed by DAGs of tasks) in an HC environment consisting of multicore chips. The goal of resource allocations is to meet the deadline constraint while being robust against uncertainty in execution times. We modeled a heterogeneous computing system used for satellite image processing, defined a mathematical robustness metric, and created and evaluated resource allocation heuristics that maximize this robustness metric. We showed that there is an important advantage in using robustness to guide the resource allocation done by the heuristics.

The full ordering of tasks generated by the heuristics (e.g., DATCP, MC DL, and HRD) is an important factor for making a good allocation of resources in the system. This is made evident by the experiment that substituted the fitness function used to assign tasks in the DATCP heuristic with the function used for the HRD. This experiment proved that there is a significant difference in how the two fitness functions performed. As computation becomes dominant over communication, the differences between heuristics that minimize makespan and those that increase robustness become less significant. This is because the makespan is more dependent on increases in the computation time.

The MC Genitor was typically unable to improve significantly the performance of the MC DL and DATCP heuristics. To observe if the performance of the MC Genitor could be improved, it was allowed to be executed longer for 5 of the 50 simulation trials. The performance, in these trials, was only improved by 1%–2%.

An interesting addition to this study would be to evaluate how these heuristics perform when we have larger systems. Another possible extension to this work would be to have a different independent variable to modify the percentage increase in communication; this would allow us to understand the effect of an increased communication time and how the heuristics perform with this change.

Acknowledgments

A preliminary version of portions of this material appeared in the 19th Heterogeneity in Computing Workshop. The authors thank Dalton Young for his valuable contributions.

References

[1] S. Ali, T.D. Braun, H.J. Siegel, A.A. Maciejewski, N. Beck, L. Boloni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, Characterizing resource allocation heuristics for heterogeneous computing systems, in: Advances in Computers, in: Parallel, Distributed, and Pervasive Computing, vol. 63, 2005, pp. 91–128.

[2] S. Ali, A.A. Maciejewski, H.J. Siegel, Perspectives on robust resource allocation for heterogeneous parallel systems, in: S. Rajasekaran, J. Reif (Eds.), Handbook of Parallel Computing: Models, Algorithms, and Applications, Chapman & Hall/CRC Press, Boca Raton, FL, 2008, 41-1–41-30.

[3] S. Ali, A.A. Maciejewski, H.J. Siegel, J.-K. Kim, Measuring the robustness of a resource allocation, IEEE Trans. Parallel Distrib. Syst. 15 (7) (2004) 630–641.

[4] S. Ali, H.J. Siegel, M. Maheswaran, D. Hensgen, S. Ali, Representing task and machine heterogeneities for heterogeneous computing systems, Tamkang J. Sci. Eng. 3 (3) (2000) 195–207. Special 50th Anniversary Issue.

[5] H. Barada, S.M. Sait, N. Baig, Task matching and scheduling in heterogeneous systems using simulated evolution, in: 10th IEEE Heterogeneous Computing Workshop, HCW '01, 2001, pp. 875–882.

[6] T.D. Braun, H.J. Siegel, N. Beck, L. Boloni, 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, J. Parallel Distrib. Comput. 61 (6) (2001) 810–837.

[7] L.-C. Canon, E. Jeannot, Evaluation and optimization of the robustness of dag schedules in heterogeneous environments, Trans. Parallel Distrib. Syst. 21 (4) (2010) 532–546.
[8] E.G. Coffman, Computer and Job-Shop Scheduling Theory, John Wiley and Sons, New York, NY, 1976.
[9] M.K. Dhodhi, I. Ahmad, A. Yatama, An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems, J. Parallel Distrib. Comput. 62 (9) (2002) 1338–1361.
[10] DigitalGlobe, Satellite imagery and geospatial information products. URL http://www.DigitalGlobe.com/.
[11] D. Fernandez-Baca, Allocating modules to processors in a distributed system, IEEE Trans. Softw. Eng. 15 (11) (1989) 1427–1436.
[12] A. Ghafoor, J. Yang, A distributed heterogeneous supercomputing management system, IEEE Comput. 26 (6) (1993) 78–86.
[13] T. Hagras, J. Janecek, A high performance, low complexity algorithm for compile time job scheduling in homogeneous computing environments, Parallel Comput. 31 (7) (2005) 653–670.
[14] O.H. Ibarra, C.E. Kim, Heuristic algorithms for scheduling independent tasks on non-identical processors, J. ACM 24 (2) (1977) 280–289.
[15] M. Kafil, I. Ahmad, Optimal task assignment in heterogeneous distributed computing systems, IEEE Concurr. 6 (3) (1998) 42–51.
[16] A. Khokhar, V.K. Prasanna, M.E. Shaaban, C. Wang, Heterogeneous computing: challenges and opportunities, IEEE Comput. 26 (6) (1993) 18–27.
[17] S. Kim, S. Lee, J. Hahm, Push–pull: deterministic search-based DAG scheduling for heterogeneous cluster systems, IEEE Trans. Parallel Distrib. Syst. 18 (11) (2007) 1489–1502.
[18] Y. Kwok, I. Ahmad, Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors, IEEE Trans. Parallel Distrib. Syst. 7 (5) (1996) 506–521.
[19] G. Liu, K. Poh, M. Xie, Iterative list scheduling for heterogeneous computing, J. Parallel Distrib. Comput. 65 (5) (2005) 654–665.
[20] A. Mahjoub, J. Pecero Sanchez, D. Trystram, Scheduling with uncertainties on new computing platforms, Comput. Optim. Appl. 48 (2) (2011) 369–398.
[21] NCAR, The national center for atmospheric research. URL http://www.ncar.ucar.edu/.
[22] V. Shestak, E.K.P. Chong, H.J. Siegel, A.A. Maciejewski, L. Benmohamed, I.-J. Wang, R. Daley, A hybrid branch-and-bound and evolutionary approach for allocating strings of applications to heterogeneous distributed computing systems, J. Parallel Distrib. Comput. 28 (3) (2008) 1157–1173.
[23] V. Shestak, J. Smith, H.J. Siegel, A.A. Maciejewski, Stochastic robustness metric and its use for static resource allocations, J. Parallel Distrib. Comput. 68 (8) (2008) 1157–1173.
[24] S. Shivle, H.J. Siegel, A.A. Maciejewski, P. Sugavanam, T. Banka, R. Castain, K. Chindam, S. Dussinger, P. Pichumani, P. Satyasekaran, W. Saylor, D. Sendek, J. Sousa, J. Sridharan, J. Velazco, Static allocation of resources to communicating subtasks in a heterogeneous ad hoc grid environment, J. Parallel Distrib. Comput. 66 (4) (2006) 600–611. Special Issue on Algorithms for Wireless and Ad-hoc Networks.
[25] G. Sih, E. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Trans. Parallel Distrib. Syst. 4 (2) (1993) 175–187.
[26] H. Singh, A. Youssef, Mapping and scheduling heterogeneous task graphs using genetic algorithms, in: 5th IEEE Heterogeneous Computing Workshop, HCW 1996, 1996, pp. 86–97.
[27] J. Smith, V. Shestak, H.J. Siegel, S. Price, L. Teklits, P. Sugavanam, Robust resource allocation in a cluster based imaging system, Parallel Comput. 35 (7) (2009) 389–400.
[28] X. Tanga, K. Li, G. Liao, R. Li, List scheduling with duplication for heterogeneous computing systems, J. Parallel Distrib. Comput. 70 (4) (2010) 323–329.
[29] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274.
[30] 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, J. Parallel Distrib. Comput. 47 (1) (1997) 8–22. Special Issue on Parallel Evolutionary Computing.
[31] D. Whitley, The genitor algorithm and selective pressure: why rank based allocation of reproductive trials is best, in: 3rd International Conference on Genetic Algorithms, 1989, pp. 116–121.
[32] M. Wu, D. Gajski, Hypertool: a programming aid for message-passing systems, IEEE Trans. Parallel Distrib. Syst. 1 (3) (1990) 330–343.
[33] D. Xu, K. Nahrstedt, D. Wichadakul, QoS and contention-aware multi-resource reservation, Clust. Comput. 4 (2) (2001) 95–107.

Luis Diego Briceño obtained his Ph.D. degree in Electrical and Computer Engineering at Colorado State University, and his B.S. degree in Electrical and Electronic Engineering from the University of Costa Rica. He is currently a component design engineer at Intel. His research interests include heterogeneous parallel and distributed computing.

Jay Smith co-founded Lagrange Systems in 2012 and currently serves as the Chief Technical Officer of the Company. Jay received his Ph.D. in Electrical and Computer Engineering from Colorado State University in 2008. Jay has co-authored over 30 peer reviewed articles in the area of parallel and distributed computing systems. In addition to his academic publications, while at I.B.M., Jay received over 20 patents and numerous corporate awards for the quality of those patents. Jay left I.B.M. as a Master Inventor in 2008 to focus on High Performance Computing at DigitalGlobe. There, Jay pioneered the application of GPGPU processing within DigitalGlobe. In addition to his position at Lagrange Systems, Jay serves as a research faculty member in the Electrical and Computer Engineering Department at Colorado State University. His research interests include high performance computing and resource management. He is a member of both the IEEE and the ACM.

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.

Anthony A. Maciejewski received the BSEE, 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.

Paul Maxwell was commissioned as an Armor officer in 1992. His military assignments include Battalion XO/S-3, Brigade S-4, Company Commander, Scout Platoon Leader, Company XO, and Mech. Infantry Platoon Leader. At West Point, he has served as an instructor and assistant professor in the department of Electrical Engineering and Computer Science. He has a BSEE, MSEE, and Ph.D. in electrical engineering. His research interests include programmable logic, computer architecture, robotics, and robustness. He is a member of the Institute of Electrical and Electronics Engineers (IEEE).

Russ Wakefield has been an instructor in the Computer Science department at Colorado State University since 2007. His research areas include distributed systems, security in operating systems, and education in Computer Science. He received his B.S. and MCS in Computer Science from Colorado State University and achieved over 25 years of industry experience in high performance computing, symmetric multiprocessing systems programming and management in industry before returning to academia.

Abdulla Al-Qawasmeh currently holds the position of Lead Software Developer at step2compliance. His work is focused on web and cloud-based applications. He received the B.S. degree in Computer Science and Information Systems from Jordan University of Science and Technology in 2005, the M.S. degree in Computer Science from the University of Houston Clear Lake in 2008, and the Ph.D. degree in Computer Engineering from Colorado State University. His research interests include robust, power and energy-efficient scheduling techniques in heterogeneous computing systems and data centers, and the characterization of heterogeneous computing systems.

Ron C. Chiang received his B.S. degree in computer science and information engineering from Tamkang University, Tamsui, Taiwan, in 1999, and M.S. degree in computer science and information engineering from National Chung Cheng University, Chiayi, Taiwan, in 2001. He is currently pursuing the Ph.D. degree in electrical and computer engineering at the George Washington University, DC. He was a software engineer with the Institute for Information Industry, Taipei, Taiwan, from 2001 to 2006 and a research assistant with the Institute of Information Science, Academia Sinica, Tapei, Taiwan, from 2006 to 2008. His research interest includes virtualization technology, cloud computing, file and storage systems, and embedded systems.

Jiayin Li received the B.E. and M.E. degrees from Huazhong University of Science and Technology, China, in 2002 and 2006, respectively, and his Ph.D. in Electrical and Computer Engineering from the University of Kentucky in 2012. His research interests include software/hardware co-design for embedded system and high performance computing.