A Framework for Mapping with Resource Co-Allocation in Heterogeneous Computing Systems

Ammar H. Alhusaini * and Viktor K. Prasanna*
Department of EE-Systems, EEB 200C
University of Southern California
Los Angeles, CA 90089-2562
Ph: (213) 740-4483
{ammar + prasanna}@usc.edu

C.S. Raghavendra
The Aerospace Corporation
P. O. Box 29257
Los Angeles, CA 90009
Ph: (310) 336-1686
raghu@aero.org

Abstract

It is often the case in Heterogeneous Computing (HC) systems that an application requires multiple resources of different types to be allocated simultaneously. In general, this problem is the resource co-allocation problem. In this paper, we develop a general framework for mapping a collection of applications with resource co-allocation requirements. In our framework, application tasks have two types of constraints to be satisfied: precedence constraints and resource sharing constraints. We use a graph theoretic framework to capture these constraints. A Directed Acyclic Graph is used to represent precedence constraints of tasks within an application and a Compatibility Graph is used to represent resource sharing constraints among tasks of applications. Both these graphs are used to find maximal independent sets of tasks that can be executed concurrently.

The objective of the mapping is to minimize the overall schedule length for a given set of applications. We develop heuristic algorithms to solve the mapping problem with resource co-allocation constraints. We also provide a two-phase algorithm that can be used for run-time adaptation. We conducted extensive simulation experiments to evaluate the performance of our heuristic algorithms. Simulation results for our algorithms show a performance improvement of 10% to 30% over a baseline algorithm of list scheduling which considers only the precedence constraints and allocates tasks from the resulting order. This paper demonstrates the importance of considering the co-allocation requirements when mapping applications in heterogeneous computing environments including grid environments.

*Supported by the DARPA/ITO Quorum Program through the Naval Postgraduate School under subcontract number N62271-97-M-0931.

1. Introduction

In *Heterogeneous Computing* (HC) systems [8, 13, 20, 25, 26], a diverse set of resources are used in a coordinated and effective way to solve computationally challenging applications. Such systems are also called *metacomputing* systems [29] or *computational grids* [10]. In general, such HC systems have compute resources with different capabilities, input/output devices, data repositories, and other resources all interconnected by heterogeneous local and wide area networks. A major challenge in using HC systems is to effectively use all the available resources.

Mapping applications in HC system is a well researched problem in the literature. The mapping problem is defined as the problem of assigning application tasks to suitable resources (matching problem) and ordering task executions in time (scheduling problem) to optimize a specific objective function. Many algorithms exist for mapping applications in HC systems (for a detailed classification see [4]). For applications consisting of several tasks and represented by Directed Acyclic Graphs (DAGs), many static and dynamic mapping algorithms have been proposed. Dynamic algorithms include the Hybrid Remapper [23], the Generational algorithm [12], as well as others [1, 18, 21]. Several static algorithms for mapping application DAGs in HC systems are described in [19, 24, 27, 32]. Most of the previous algorithms focus on compute resources only.

In our earlier work [2], we developed a unified resource scheduling framework for HC systems which supports multiple resource requirements, advance reservation, and data replication. Each application was assumed to consist of several tasks and was represented by a DAG. A task's input data can be data items from its predecessors and/or data sets from data repositories. Input data sets can be accessed from one or more data repositories. Sources of input data and the execution times of the tasks on various machines along with their availability were considered simultaneously to minimize the overall completion time. Although we considered multiple resource requirements in [2], tasks were not required to access different types of resources simultaneously.

In this paper, we consider the problem of mapping a set of applications in a HC system where application tasks require *concurrent access* to multiple resources of different types. In general, this problem is the *resource co-allocation* problem. For example, an interactive data analysis application may require simultaneous access to a storage system holding a copy of the data, a supercomputer for analysis, network elements for data transfer, and a display device for interaction [11]. For such applications, co-allocation of all required resources is necessary. A special case of this problem where a single application requires concurrent access to a set of resources in a computational grid has been considered in [5].

In this paper, we develop a general framework for mapping with resource co-allocation in HC systems. The framework defines the system and application models and formulates the co-allocation problem. Two graphs are used to represent applications: a Directed Acyclic Graph (DAG) and a *Compatibility Graph* (defined in Section 3.4). DAG representation is given initially and stay unchanged throughout the mapping process while the compatibility graph is updated during the mapping process. In classical mapping problems, only DAGs are used to represent the precedence constraints among tasks. In this paper, the co-allocation requirements add another type of constraint among the tasks: the resource sharing constraint which is captured in the compatibility graph. Tasks that share one or more resources cannot be executed concurrently due to the resource sharing constraints even if they have no precedence constraints among them. Known mapping algorithms for the classical DAG scheduling problem cannot be directly used for the above problem since they only consider the precedence constraints. In this paper, we develop heuristic algorithms that can be used with different allocation techniques to efficiently solve the co-allocation problem defined by our framework.

In our approach, multiple DAGs of different applications are combined into a single DAG. All tasks that have satisfied the precedence constraints are ready for allocation provided they have no resource sharing constraints. Using the compatibility graph, we will select tasks that can be executed concurrently. This is achieved by finding maximal independent sets in the compatibility graph.

The co-allocation problem formulated here will be applicable to the MSHN project [16]. MSHN project is a collaborative effort between DoD (Naval Postgraduate School), academia (NPS, USC, Purdue University), and industry (NOEMIX). MSHN (Management System for Heterogeneous Networks) is designing and implementing a Resource Management System (RMS) for distributed heterogeneous and shared environments. MSHN assumes heterogeneity in resources, processes, and QoS requirements. Processes may have different priorities, deadlines, and compute characteristics. The goal is to schedule shared resources among individual applications so that their Quality of Service (QoS) requirements are satisfied. MSHN supports adaptive applications that can exist in several different versions. These versions may differ in the precision of computation or input data, and therefore have different values to a user. Unlike other HC and grid projects, MSHN seeks to determine how to meet QoS requirements of multiple application simultaneously.

The rest of this paper is organized as follows. In next section we give the definition of the co-allocation problem and summarize some related work. The problem framework is defined in Section 3. In Section 4, we give the outline of our approach to solve the co-allocation problem using our framework. Experimental results are given in Section 5. Finally, Section 6 gives the conclusions and future research directions.

2. The Co-Allocation Problem

The co-allocation problem can be defined as the problem of simultaneously allocating multiple resources of different types to an application in order to meet specific performance requirements. The need of co-allocation is a common characteristic of applications in HC systems (as well as computational grids). For example, an application may require a data repository, a HPC platform, multiple display devices, and communication links all to be allocated simultaneously.

Resource co-allocation has been introduced in the high-performance distributed computing community by the Globus project [5]. The co-allocation problem is defined as the provision of allocation, configuration, and monitoring/control functions for the resource ensemble required by a single application [5]. The Globus tool-kit provides a flexible set of co-allocation mechanisms that can be used to construct application-specific co-allocation strategies. Only compute resources are considered in the Globus project at this time, to synchronize the start of complex applications at multiple sites.

The notion of co-allocation was also considered in the Legion project [22]. In the Legion project, an *Enactor* provides a mechanism to co-allocate compute and storage resources (hosts and vaults) to a single application. The co-allocation mechanism is based on advance resource reservation.

In [5] and [22], the focus is on implementation issues of the co-allocation process. Algorithms for efficient mapping with co-allocation requirements are not considered. Also, both projects focus on executing a single application. The problem becomes challenging when a number of applications share resources.

In this paper, we study the co-allocation problem in the context of mapping a set of applications where each application is represented by a DAG. We consider conflicts among tasks caused by precedence constraints as well as due to resource sharing. The objective is to minimize the overall schedule length for a set of applications. Our main contribution in this paper is the formulation of the mapping problem in the presence of co-allocation requirements for multiple applications. To the best of our knowledge, this work is the first step towards a general framework for mapping applications with resource co-allocation in HC systems.

3. The Framework

3.1 System Model

We consider a heterogeneous computing system with m compute resources (machines), $M = \{m_1, m_2, \ldots, m_m\}$, and a set of r resources, $R = \{r_1, r_2, \ldots, r_r\}$. Compute resources can be HPC platforms, workstations, personal computers, etc. Resource $r_k \in R$ can be a data repository, an input/output device, etc. We assume that only one task can use any resource (compute and non-compute resource) at any given time. Resources are interconnected by heterogeneous communication links. Communication costs are given by two matrices: MM_comm and RM_comm, where MM_comm gives the communication cost for transferring a byte between machines and RM_comm gives the communication cost for transferring a byte between the resources in R and the machines.

We assume that an estimate of the computation time of a given task t_i on machine m_j is available at compile-time. These estimated computation times are given in an *Estimated Computation Time* (ECT) matrix. Thus, $ECT(t_i, m_j)$ gives the estimated computation time for task t_i on machine m_j. If task t_i cannot be executed on machine m_j, then $ECT(t_i, m_j)$ is set to infinity.

$MA(m_j)$ gives the earliest time when machine m_j is available and $RA(r_k)$ gives the earliest time when resource r_k is available. As the mapping proceeds, the earliest time when a resource (m_j or r_k) is available is calculated as the finish time of the last task assigned to this resource.

3.2. Application Model

In this HC system, a set of N applications, $A = \{A_1, \ldots, A_N\}$, compete for system resources. Each submitted application consists of several tasks and is modeled by a DAG, where the nodes represent computational requirements and the edges represent both precedence constraints and communication requirements. Figure 1 shows an example of two application DAGs. We assume that the whole set of applications to be mapped is known *apriori* (static applications). The problem is to execute these N applications as efficiently as possible. Our approach is to combine all submitted application DAGs into a single DAG, $G = (T, E)$, where T represents the set of tasks to be executed from all applications, $T = \{t_1, t_2, \ldots, t_n\}$, and E represents the data dependencies and communication between tasks. Edge e_{ij} indicates that there is communication from task t_i to t_j and its weight denotes the amount of communication. G is constructed by connecting the root nodes (tasks) of all applications to a hypothetical zero-cost entry node with zero-weight edges.

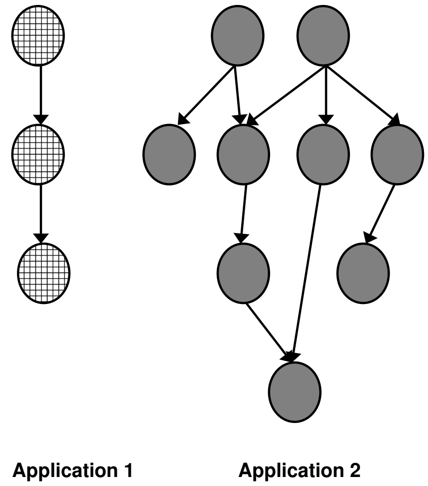

Figure 1. An example of two application DAGs

We assume that each task t_i needs *concurrent access* to a set of resources: one compute resource m_j and a number of additional resources as specified by the set $R(t_i)$, where $R(t_i) \subseteq R$. The amount of data to be transferred between t_i and r_k, where $r_k \in R(t_i)$, is given by $DATA(t_i, r_k)$. A task t_i cannot start execution until all its required resources are available to the task. All required resources will be allocated to the task during its execution. These resources will be available after the task completes its execution. We assume that all required resources are acquired and released at the same time (atomic transaction).

We say that task t_i and task t_j are *incompatible* if and only if $R(t_i) \cap R(t_j) \neq \phi$. Incompatible tasks cannot be executed concurrently even if they have no precedence constrains among them. Therefore, in our framework, tasks may be unable to run concurrently for either of the following reasons: (1) precedence constraints, or (2) resource sharing constraints.

The execution time of task t_i on machine m_j, $Exec(t_i, m_j)$, depends on the computation time of t_i on m_j and data transfer times between m_j and all resources which t_i needs to access during its execution. For example, for systems that assume computation and communication cannot be overlapped, $Exec(t_i, m_j)$ can be defined as

$$Exec(t_i, m_j) = ECT(t_i, m_j) + \sum_{r_k \in R(t_i)} (DATA(t_i, r_k) \times RM\_comm(r_k, m_j))$$

where the last term gives the total time to transfer any required data between machine m_j and every resource $r_k \in R(t_i)$. $Exec(t_i, m_j)$ can also be defined in different ways to consider the overlapping between computation and communication as well as other communication models.

The average execution time of task t_i is defined as

$$\overline{Exec(t_i)} = \sum_{j=1}^{m} Exec(t_i, m_j)/m$$

$ST(t_i, m_j)$ and $FT(t_i, m_j)$ are the earliest *start time* and the earliest *finish time* of task t_i on machine m_j, respectively if t_i were to be mapped on m_j. $ST(t_i, m_j)$ is defined as

$$ST(t_i, m_j) = \max\{MA(m_j), Data\_Pred(t_i, m_j)\}$$

where $Data\_Pred(t_i, m_j)$ is the time when task t_i receives all the needed data from all tasks in its predecessor set, $Pre(t_i)$, if t_i is mapped onto machine m_j. $FT(t_i, m_j)$ is defined as

$$FT(t_i, m_j) = ST(t_i, m_j) + Exec(t_i, m_j)$$

3.3. Mapping Objective

The objective function in our framework is to determine an assignment (matching) of tasks to compute resources and schedule their executions based on all resource requirements such that the overall schedule length (or *makespan*) of all submitted applications is minimized while satisfying all

1. Application-specified precedence constraints and
2. Implied resource sharing constraints.

Thus, we can define our objective function as

$$Minimize \ \{\max_{i=1}^{N} [Finish\ Time(A_i)]\},$$

where $Finish\ Time(A_i)$ is the completion time of application A_i. Note that the resource sharing constraint is a dynamic constraint - it depends on tasks ready to be allocated and their resource requirements.

Task	Resource Requirements
V_1	r_1, r_2
V_2	r_2, r_3
V_3	r_3, r_5
V_4	r_1, r_4
V_5	r_4, r_5, r_6
V_6	r_6

Table 1. An example showing 6 tasks and their resource requirements

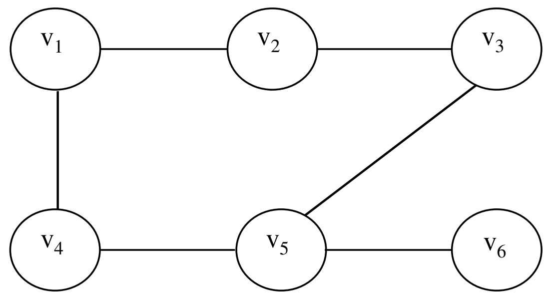

Figure 2. The compatibility graph for the tasks shown in Table 1

Task	Execution Time
V_1	5
V_2	6
V_3	2
V_4	4
V_5	1
V_6	3

Table 2. Execution times for the tasks in Figure 2

3.4. Compatibility Graph

To capture the implied resource sharing constraints among tasks that may belong to the same or different applications, we use the *compatibility graph*, $g = (V, E)$, where vertex v_i denotes task t_i and edge e_{ij} exists if and only if t_i and t_j are incompatible. Recall, task t_i and task t_j are incompatible if and only if $R(t_i) \cap R(t_j) \neq \phi$. An *independent set* [6] is a set of vertices of g such that no two vertices of the set are adjacent. An independent set is called a *maximal independent set* if there is no other independent set of g that contains it. A maximal independent set with the largest number of vertices among all maximal independent sets is called a *maximum independent* set [6]. The maximum independent set problem is NP-complete [15]. In our model, a maximal independent set of g represents a maximal set of tasks that can be executed concurrently if there is no precedence constraints among them.

As an example, consider a set of 6 independent tasks. Each task needs concurrent access to a set of resources as specified in Table 1. The compatibility graph g for this example is shown in Figure 2. The maximal independent sets of g are $\{V_1, V_5\}$, $\{V_2, V_5\}$, $\{V_1, V_3, V_6\}$, $\{V_2, V_4, V_6\}$, and $\{V_3, V_4, V_6\}$. The last three sets are maximum independent sets.

4. Our Solution Approach

In classical DAG scheduling problem, application DAGs are partitioned onto levels such that each level contains independent tasks, i.e., there are no data dependencies among the tasks in the same level. Therefore, all tasks in the same level can be executed concurrently. In our framework, incompatible tasks in the same level cannot be executed concurrently due to resource sharing constraints. Therefore, mapping algorithms for the classical DAG scheduling problem (ex. [1, 30, 18, 23, 9, 31, 32]) cannot be directly used for our problem.

In this section, we develop a static algorithm using the framework defined in Section 3. The algorithm can be used with different heuristics and strategies to obtain efficient solutions to the mapping problem with co-allocation requirements. Also, we provide a two-phase algorithm that can be used for run-time adaptation.

4.1. The Co-Allocation Algorithm

Pseudo code for our co-allocation algorithm is given in Figure 3. With the given set of applications and resource requirements of tasks, we first find tasks that have satisfied precedence constraints and then select maximal independent sets among these for allocation. The compatibility graph is used to find maximal independent sets. Since the maximum independent set problem is NP-complete [15], our approach for selecting a maximal independent set is based on first choosing a critical node v_c, and then finding a maximal independent set that contains v_c. Different strategies for selecting critical nodes are given in Section 4.2.

To ensure precedence constrains are satisfied, we combine all submitted applications into a single DAG, G, by using zero-weight edges to connect the root nodes (tasks) of all applications to a hypothetical zero-cost entry node. Then we partition the combined DAG onto l levels such that level 0 contains the entry node and level 1 contains all tasks that do not have any predecessors in the submitted DAGs. All tasks in level l have no successors. For each task t_i in level k, all of its predecessors are in levels 0 to $k-1$, and at least one of them in level k-1.

Let RDY be the set of tasks that have no precedence constraints among them and that are ready for allocation. A task is ready for allocation if for each predecessor all required resources have been allocated. Let W be the set of ready tasks that are waiting for allocation, and $ALLOCATED$ be the set of allocated tasks. The list $SCHEDULE$ will give the resulting scheduling order of tasks.

In steps 1 and 2 of the algorithm, we combine all submitted applications into a single DAG, G, and partition G into l levels. Then the algorithm proceeds level-by-level as follows. For each level l of G, we construct the compatibility graph g for all tasks in this level (step 5). g is used to find maximal independent sets of tasks that can be executed concurrently. The first maximal independent set of tasks to be allocated is selected in steps 6-7 where a critical node v_c is selected in step 6 and a maximal independent set that contains v_c is selected in step 7.

In step 9, all tasks in the selected maximal independent set are allocated to their required resources. For the allocation, we first find the scheduling order of the tasks using different heuristics that are given in Section 4.3. Then, we use this scheduling order to assign a compute resource m_j to each task t_i in order to minimize its finish time $FT(t_i, m_j)$. Availability times ($MA(m_j)$ and $RA(r_k)$) of all resources required by task t_i are updated based on $FT(t_i, m_j)$.

In steps 11-15, a new set of maximal independent tasks among all waiting tasks is selected to be allocated at next allocation event. The next allocation event is calculated as the earliest finish time, $FT(t_i, m_j)$, among all allocated tasks. An allocated task v_x with the earliest finish time is identified in step 11 and then removed from the $ALLOCATED$ set. Initially (as in step 13), the set of candidate tasks that can be allocated next ,C, contains all waiting tasks. The candidate set ,C, is updated in step 14 by removing all tasks that are incompatible with any allocated task. Then g is used to find a maximal independent set of tasks from C. The algorithm repeats steps 9-15 until all tasks in this level have been allo-

Inputs: application DAGs, estimated computation and communication costs, and resource requirements of tasks
Outputs: the scheduling order of tasks ($SCHEDULE$) and the schedule length based on the given inputs

Begin
1. Combine all submitted application DAGs into a single DAG (G)
2. Do level partitioning of G /* tasks in each level have no presedence constraints */
3. For level 1 to l do
4. Initialize W to include all tasks in the current level and let $ALLOCATED = \phi$
5. Construct the compatibility graph g for all tasks in the current level
6. Pick a critical node v_c from W /* several strategies can be used for critical node selection*/
7. Find a maximal independent set of tasks S from W such that $v_c \in S$ /* g is used to find the maximal independent set */
8. While W is not empty do
9. Allocate all tasks in S to their required resources in the following way:
9a. Find the scheduling order of the tasks and add them to $SCHEDULE$ /* different heuristics can be used */
9b. For each task t_i in S (in the scheduling order) do
- Assign a compute resource m_j to task t_i in order to minimize its finish time $FT(t_i, m_j)$
- Update $MA(m_j)$ and $RA(r_k)$, $\forall r_k \in R(t_i)$, based on $FT(t_i, m_j)$
10. Add all tasks in S to $ALLOCATED$ and remove them from W
11. Let v_x be the allocated task with the lowest finish time
12. Remove v_x from $ALLOCATED$
13. Let $C = W$, where C is the set of candidate tasks that can be allocated next
14. Remove all tasks from C that are incompatible with any allocated task
15. If ($C \neq \phi$)
15a. Pick a critical node v_c from C such that v_c is adjacent to v_x in g
15b. Find a maximal independent set of tasks S from C such that $v_c \in S$
16. End (while)
17. End (for)
End

Figure 3. The co-allocation algorithm

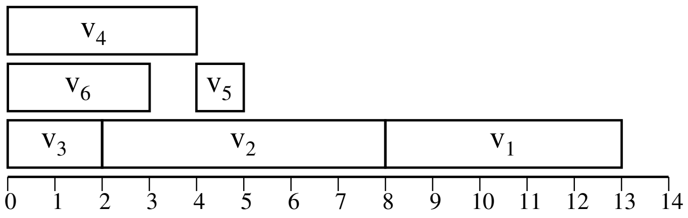

Figure 4. Schedule 1

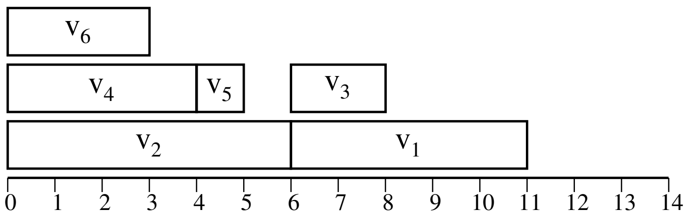

Figure 5. Schedule 2

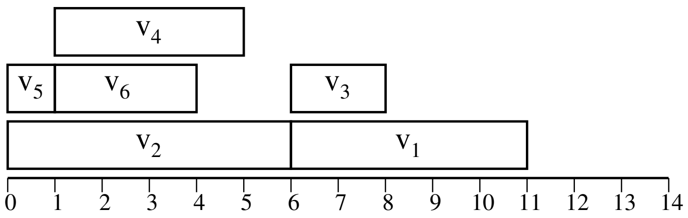

Figure 6. Schedule 3

cated.

4.2. Maximal Independent Set Selection

Our approach for selecting a maximal independent set is based on first choosing a critical node v_c, and then finding a maximal independent set that contains v_c. Critical nodes need to be selected carefully.

The length of the schedule is influenced by the selection of maximal independent sets. This is shown in the following example using the compatibility graph in Figure 2. For this example, for the sake of simplicity, we assume that all the resource requirements of tasks (compute and non-compute resources) are pre-specified. Therefore, the execution times of all tasks are known *apriori*. These times are shown in Table 2. Example schedules are given in Figures 4-6. These schedules have different schedule lengths. The optimal length of the schedule is 11 time units. This is achieved by schedules 2 and 3. In schedules 1 and 2, two different maximum independent sets were selected to be scheduled first. Schedule 1 has a length of 13 time units, while schedule 2 has the optimal length. This clearly shows the importance of the order in which the maximal independent sets are considered for scheduling. From schedule 3, we can also see that it is not always efficient to select a maximum independent set to be scheduled first. Schedule 3, which starts with a maximal independent set $\{V_2, V_5\}$ (not a maximum), has the optimal length while schedule 1, which starts with a maximum set $\{V_3, V_4, V_6\}$, has a non-optimal length of 13 time units.

Since the maximum independent set problem is NP-complete [15], we will use different heuristic techniques for selecting a maximal independent set S. The idea behind our techniques is to select a *critical* vertex v_c and add it to S which is initially empty. Then we attempt to enlarge S by traversing g. Different strategies can be used for selecting critical vertices. In the following we describe some of these strategies.

S1 Highest average execution time. In this strategy, we give priority for tasks that need more time for execution since they can be critical. In HC systems, tasks have different execution times on different machines. Therefore, we use the average execution time $\overline{Exec(t_i)}$ as the selection criterion.

S2 Highest degree. The node out-degree in a DAG has been used in many list scheduling heuristics as a priority function. The out-degree of a node t_i gives the number of tasks that have precedence constraints with t_i. The idea is to advance the execution of tasks with high out-degree. Thus, many tasks will be ready for mapping once high out-degree tasks complete execution. In our framework, the out-degree of node t_i in the combined DAG G (which represents task t_i) is not reflecting all dependencies between t_i and other tasks since G only captures the precedence constraints. Resource sharing dependencies should also be considered. Therefore, we define the degree of task t_i as the sum of its out-degree in G and its degree in g. This number gives the total number of tasks that will be ready for mapping once t_i completes its execution, either because those tasks have a precedence or resource sharing dependencies with t_i.

S3 Critical path nodes. A Critical Path (CP) in a DAG is a path from an entry node to an exit node with the largest finishing time. We use average execution times and average communication costs to find the critical path. In some situations, the average execution time or the degree of a task t_i cannot reflect how important for other tasks that t_i finish execution as soon as possible. The successors of t_i may not be critical tasks and advancing their execution may not improve the schedule length. For these reasons, selecting critical path nodes from G as critical tasks can be a good strategy. In this paper, we implement two versions of this strategy:

S3.1 In this version, the task that is on the critical path is selected as a critical task. If there is no such task among the current set of candidate tasks, the task with the highest average execution time is selected as a critical task.

S3.2 This version is similar to the first one except for the case when there is no critical path node among the current set of candidate task. In this case, the task with the highest degree is selected as a critical task.

S4 Maximum weighted clique. In [3], a similar approach to the compatibility graph has been used for scheduling independent tasks. Each task in [3] requires a simultaneous access to a set of pre-specified processors. All resource (processor) requirements and all execution times were assumed to be known *apriori*. It has been shown in [3] that the weight of the maximum weighted clique in the *constraint graph* (compatibility graph) is a lower bound on the optimum makespan, where the weight of each node is its execution time. In our previous example, we can notice that the weight of the maximum weighted clique (V_1 and V_2) is 11 and the optimal schedule length is 11. We can also notice that any selected maximal independent set should contain a task that belong to the maximum weighted clique in order to achieve the optimal schedule length. Inspired by this observation, we can use maximum weighted clique nodes as candidates for selecting critical tasks. Since we assume that not all required resources are pre-specified and, thus, the actual execution times cannot be known *apriori*, we will use the average execution times as the nodes' weights. It is obvious that in our model, maximum weighted clique cannot guarantee the optimal solution but it could be a good heuristic for selecting maximal independent sets.

4.3. Allocation Heuristics

After selecting a maximal independent set of tasks, careful allocation of these tasks to compute resources (machines) is required to achieve our objective. Different heuristic techniques can be used for allocating tasks of the selected maximal independent set S to compute resources. In the following we describe some of our allocation heuristics. The idea behind our heuristics is to advance the execution of tasks that may be critical in order to minimized the overall schedule length.

1. **Highest Average-Execution-Time First (HAETF)**. In this heuristic, the average execution time is used as a priority function to place tasks in a list. All tasks will be placed in a list in the order of non-increasing average execution times. Using this order, each task will be allocated to the required resources such that its finish time is minimized.

2. **Maximum Finish-Time First (MAX)**. For each task, we calculate the best finish time that can be achieved.Then we select the task with the maximum best finish time among all tasks. The task will be allocated to its required resources such that its finish time is minimized. We repeat until all tasks are allocated.

3. **Minimum Finish-Time First (MIN)**. This heuristic is similar to the Maximum Finish-Time First (MAX) heuristic except that we select the task with the minimum finish time instead of selecting the task with the maximum finish time.

4. **Highest Degree First (HDF)**. In this heuristic, all tasks will be placed in a list in descending order according to their degrees. Then, tasks will be allocated one-by-one to the required resources such that the finish time for each task is minimized.

4.4. Two-Phase Algorithm

We propose a two-phase algorithm for run-time adaptation using our static co-allocation algorithm. The two-phase algorithm can be used for the problem of mapping with resource co-allocation as defined in our framework as follows.

1. **Compile-time mapping**. At this phase, the co-allocation algorithm described in Section 4.1 is used to obtain an ordered list of tasks. The order of tasks in the list is based on their scheduling order as produced by our co-allocation algorithm. The list is obtained by satisfying all precedence and resource sharing constraints with the objective of minimizing the overall schedule length. Estimated computation and communication times are used to calculate the schedule length.

2. **Run-time Adaptation**. Run-time adaptation is needed for the cases when actual execution times differ from estimated execution times. One way to consider this is to scan through the ordered list obtained in phase one once a task completes its execution in order to find all tasks that can be executed at this time. The scanning can be done through a window of tasks with specific size k, where $k > 0$.

4.5. Implementation Issues

The focus of this paper is the mapping problem with resource co-allocation requirements in HC systems. The implementation details for the co-allocation process are outside the scope of this paper. A good discussion about many issues in the co-allocation process implementation can be found in [5]. In the following we state our assumptions in this paper regarding the co-allocation implementation.

We assume that a task t_i cannot start execution until all its required resources are available. These resources will be acquired at the same time. Once a task t_i completes its execution, all its allocated resources will be released at the same time and will be available for other tasks. We assume that any allocation request for any resource will be granted as long as this resource is available. We do not consider the cases of resource failures that can occur in the HC and Grid environments. The issue of failure modes in the co-allocation process and other implementation details can be found in [5].

5. Performance Evaluation

A simulator was implemented to evaluate the performance of our co-allocation algorithms and the proposed selection strategies and allocation heuristics discussed in Section 4. In this section, we explain our simulation procedure and give experimental results.

5.1. Simulation Procedure

To define the HC system, numbers of machines and resources are given to the simulator as inputs. Communication costs among all resources are selected randomly from a uniform distribution with a mean equal to *ave_comm*. The communication costs are source and destination dependent.

The workload consists of randomly generated DAGs. Random DAGs are generated as follows: The number of tasks in the graph, *no_tasks*, maximum out-degree of a node, *max_outdegree*, average computation cost of a node, *ave_comp*, and average message size to be transferred among tasks, *ave_msg_size*, are given as inputs. First, the computation time of each task on every compute resource is randomly selected from a uniform distribution with a mean equal to *avg_comp*. Starting with the first task, the number of children (out-degree) is randomly selected between 1 and *max_outdegree*. Then, children are randomly selected for this task. The weight of each edge in the DAG is randomly selected from a uniform distribution with a mean equal to *ave_msg_size*. Resource requirements for each task are randomly selected from available resources. The amount of data to be transferred to/from each resource in the resource requirements set is randomly selected from a uniform distribution with a mean equal *ave_data_size*. The sizes of random DAGs range from 50 to 250 tasks with increments of 50.

5.2. The Baseline Algorithm

Many mapping algorithms exist in the literature for mapping DAGs in HC systems. None of these algorithms consider the co-allocation problem we define in this paper. Therefore, we will use a simple list scheduling algorithm as a baseline algorithm to evaluate our co-allocation algorithm. The baseline algorithm is a fast static algorithm for mapping DAGs in HC environments. It partitions the tasks in the DAG into levels using an algorithm similar to the level partitioning algorithm described in Section 4.1. Then all the tasks are ordered such that the tasks in level k come before the tasks in level $k+1$. The tasks in the same level are sorted in descending order based on the average execution time of each task (ties are broken arbitrarily). The tasks are considered for mapping in this order. A task is mapped to the required resources such that its finish time is minimized.

The Baseline algorithm is similar to our algorithms in the sense that all algorithms proceed level-by-level. In the Baseline algorithm, the scheduling order of tasks at the same level is based on average execution times of tasks. On the other hand, the scheduling order in our algorithms is based on the selection of maximal independent sets. We construct the compatibility graph for each level and use it for selecting maximal independent sets. Different heuristics are used to order tasks within selected sets.

5.3. Experimental Results

Our experimental results are given in Figures 7- 14. The total number of tasks were varied from 50 to 250 with increments of 50. Each point in the figures is an average of

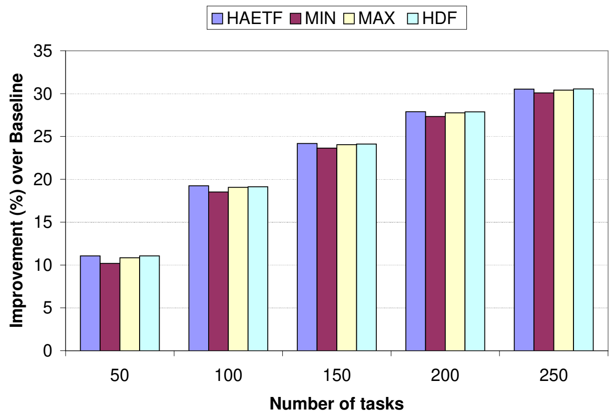

Figure 7. Performance of the allocation heuristics when using selection strategy $S1$

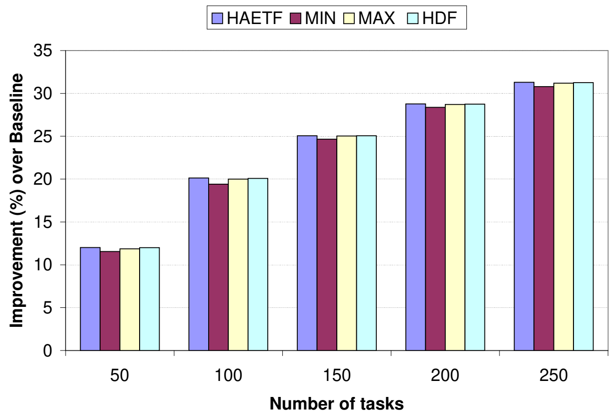

Figure 8. Performance of the allocation heuristics when using selection strategy $S2$

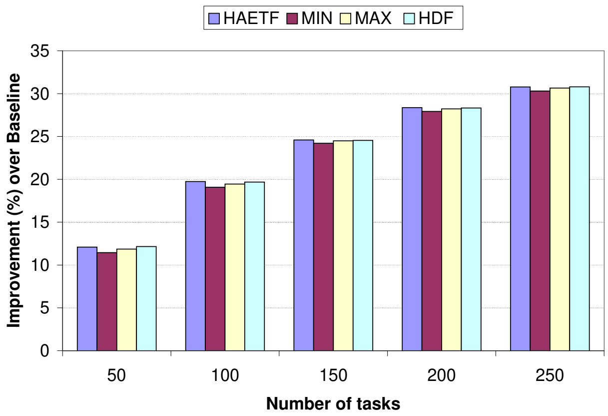

Figure 9. Performance of the allocation heuristics when using selection strategy $S3.1$

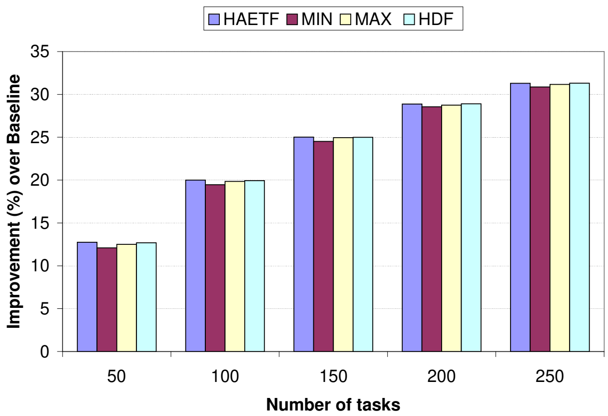

Figure 10. Performance of the allocation heuristics when using selection strategy $S3.2$

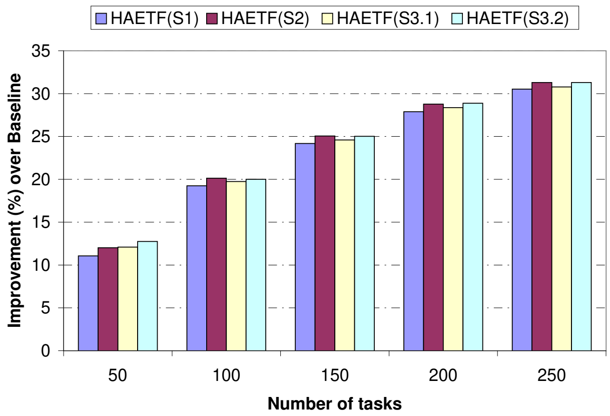

Figure 11. Performance of the selection strategies with $HAETF$ allocation heuristic

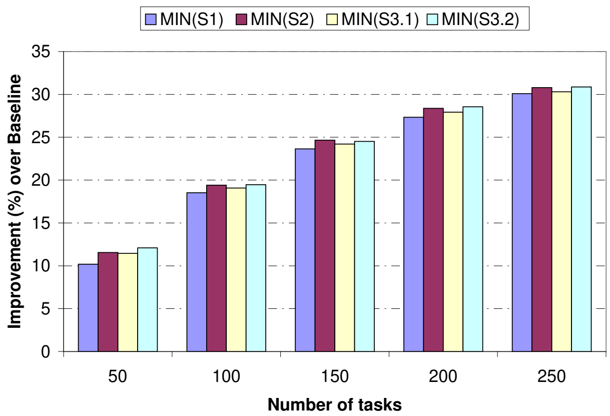

Figure 12. Performance of the selection strategies with MIN allocation heuristic

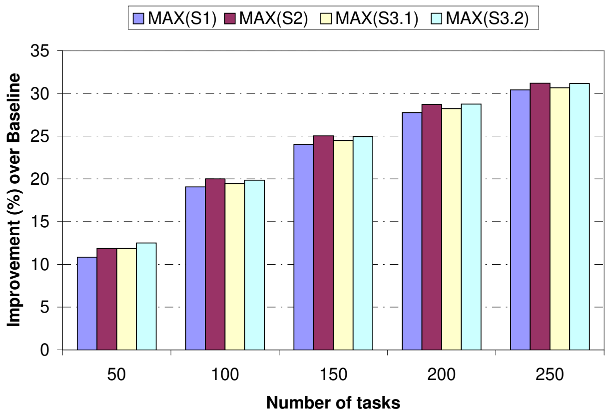

Figure 13. Performance of the selection strategies with MAX allocation heuristic

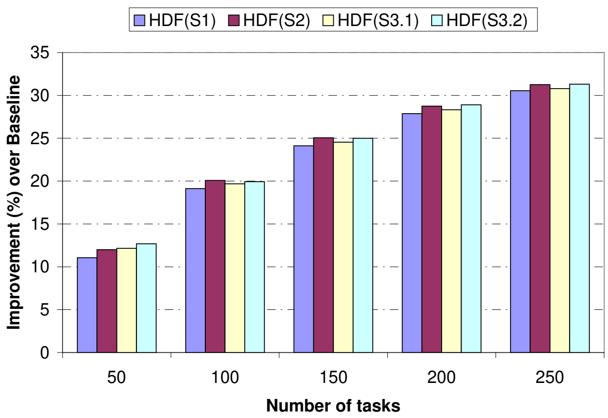

Figure 14. Performance of the selection strategies with HDF allocation heuristic

400 runs with different random DAGs. Random DAGs were generated with *max_outdegree*={2,3,4,5}.

Figures 7- 10 show the performance results of our allocation heuristics compared to the Baseline algorithm when using different maximal independent set selection strategies. The improvement of our heuristics over the Baseline increases as the total number of tasks increases. This shows the importance of considering co-allocation requirements in mapping algorithms. Generally, our allocation heuristics have relatively the same performance with a difference of less than 1%.

The performance results of maximal independent sets selection strategies when using different allocation heuristics are given in Figures 11- 14. As in the previous set of results, the improvement over Baseline algorithm increases as total number of tasks increases. Also, the selection strategies have relatively same performance with a difference less than 2%.

In our simulation study, we found that the number of machines and the number of resources did not have a significant impact on the performance of allocation heuristics and selection strategies.

6. Conclusions and Future Work

This paper proposes a novel framework for the problem of mapping applications with resource co-allocation in HC systems. We formulate the co-allocation problem and develop efficient algorithms for solving this problem using a graph theoritc approach. Our simulation results show the importance of considering the co-allocation requirements at mapping decisions.

In solving our co-allocation problem, we need to find maximal independent sets among tasks competing for system resources. Although we considered many heuristics, they all seem to perform equally indicating that a simple heuristic will suffice.

In our future work, we plan to expand our framework to consider concurrent usage of multiple compute resources and *advance resource reservations*. With advance reservation, system resources can be reserved in advanced for specific time intervals. Therefore, resource availability must be expressed as a list of available time slots and mapping algorithms should be insertion-based algorithms. To co-allocate a set of resources in this case, efficient algorithms are needed to find the best time slot when all resources are available for the required duration. In this paper, our algorithms are non insertion-based since the earliest available time for a resource r_i is the finish time of the last task assigned to r_i.

References

[1] Ishfaq Ahmad and Yu-Kwong Kwok, "On parallelizing the

multiprocessor scheduling problem," *IEEE Trans. on Parallel and Distributed Systems,* 10(4):414-432, Aprill 1999.
[2] Ammar H. Alhusaini, Viktor K Prasanna, and C.S. Raghavendra, "A unified resource scheduling framework for heterogeneous computing environments,' ' *8th Heterogeneous Computing Workshop (HCW' 99),* pp. 156-165, April 12, 1999.
[3] L. Bianco, P. Dell'Olmo, and M. Grazia Sperenza, "Nonpreemptive scheduling of independent tasks with prespecified processor allocations," *Naval Research Logistics,* 41:959-971, 1994.
[4] T. Braun, H.J. Siegel, Noah Beck, Ladislau Boloni, Muthucumaru Maheswaran, Alber t Reuther, James Robertson, Mitchell Theys, and Bin Yao, "A Taxonomy for describing matching and scheduling heuristics for mixed-machines heterogeneous computing systems," *Workshop on Advances in Parallel and Distributed Systems (APADS),* West Lafayette, IN, Oct. 1998.
[5] K. Czajkowski, I. Foster, and C. Kesselman, "Resource coallocation in computational grids," *7th IEEE Symposium on High Performance Distributed Computing,* pp. 219-228, 1999.
[6] Nicos Christofides, *Graph theory: An algorithmic approach,* Academic Press, 1975.
[7] Legion Web Page. *http://legion.virginia.edu*
[8] Mary M. Eshagian (ed.), *Heterogeneous computing,* Norwood, MA: Artech House, 1996.
[9] Mary M. Eshagian and Y.C. Wu, "Mapping heterogeneous task graphs onto heterogeneous system graphs," *6th Heterogeneous Computing Workshop (HCW' 97),* pp. 147-160, 1999.
[10] I. Foster and C. Kesselman, ed., *The Grid: blueprint for new computing infrastructure,* Morgan Kaufmann Publishers, San Francisco, CA, 1999.
[11] I. Foster, C. Kesselman, Craig Lee, Bob Lindell, K. Nahrstedt, and Alain Roy, "A distributed resource management architecture that support advance reservations a nd coallocation," *Intl. Workshop on Quality of Service,* 1999.
[12] R. Freund, B. Carter, D. Watson, E. Keith, and F. Mirabile, "Generational scheduling for heterogeneous computing systems," *Int'l Conf. Parallel and Distributed Processing Techniques and Applications (PD PTA '96),* pp. 769-778, Aug. 1996.
[13] R. Freund and H. J. Siegel, "Guest editors' introduction:Heterogeneous processing" *IEEE Computer,* 26(6):13-17, June 1993.
[14] Globus Web Page. *http://www.globus.org.*
[15] Michael R. Gary and David S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness,* W.H. Freeman and Company, San Francisco, CA, 1979.
[16] D. Hensgen, T. Kidd, D. St.John, M. Schnaidt, H.J. Siegel, T. Braun, M. Maheswaran, S. Ali, J. Kim, C. Irvine, Tim Levin, R. Freund, M. Kussow, M. Godfrey, A. Duman, P. Carff, S. Kidd, Viktor Prasanna, P. Bhat, and Ammar Alhusaini, "An overview of MSHN: the Management System for Heterogeneous Networks," *8th Heterogeneous Computing Workshop (HCW' 99),* pp. 184-198, April 12, 1999.
[17] O. Ibarra and C. Kim, "Heuristic algorithms for scheduling independent tasks on non identical processors. " *Journal of The ACM,* 24(2):280-289, April 1977.
[18] M. Iverson and F. Ozguner, "Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous en vironment," *7th Heterogeneous Computing Workshop (HCW' 98),* pp. 70-78, March 1998.
[19] M. Iverson, F. Ozguner, and G. J. Follen, "Parallelizing existing applications in a distributed heterogeneous environment," *4th Heterogeneous Computing Workshop (HCW' 95),* pp. 93-100, Apr. 1995.
[20] A. Khokhar, V. K. Prasanna, M. Shaaban, and C. L. Wang, "Heterogeneous computing: challenges and opportunities," *IEEE Computer,* 26(6):18-27, June 1993.
[21] C. Leangsuksun, J. Potter, and S. Scott, "Dynamic task mapping algorithms for a distributed heterogeneous computing environm ent," *4th Heterogeneous Computing Workshop (HCW' 95),* pp. 30-34, Apr. 1995.
[22] Legion Web Page. *http://legion.virginia.edu.*
[23] M. Maheswaran and H. J. Siegel, "A Dynamic matching and scheduling algorithm for heterogeneous computing systems," *7th Heterogeneous Computing Workshop (HCW '98),* pp. 57-69,March 1998.
[24] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund, "Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous e nvironment," *5th Heterogeneous Computing Workshop (HCW' 96),* pp. 98-117, Apr. 1996.
[25] H.J. Siegel, John K. Antonio, Richard Metzger, min Tan, and Yan Alexandel Li, " Heterogeneous computing," in *Parallel and distributed computing handbook,* A.Y. Zomaya (ed.), McGraw-Hill, New york, 1996, pp.725-761.
[26] H.J. Siegel, Muthucumaru Mahesewaran, and Tracy D. Braun, "Heterogeneous distributed computing," in *Encyclopedia of electrical and electronics engineering,* J. Webster (ed.), John Wiley & Sons, New York, to appear.
[27] G. C. Sih and E. A. Lee, "A Compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans.on Parallel and Distributed Systems,* 4(2):175-187, Feb. 1993.
[28] Harmel Singh and Abdou Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," *5th Heterogeneous Computing Workshop (HCW' 96),* pp. 86-97 , April 15-16, 1996.
[29] L. Smarr and C. E. Catlett, "Metacomputing," *Communications of the ACM,* 35(6):45-52, June 1994.
[30] Haluk Topcuoglu, Salim Hariri, and Min-You Wu, "Task scheduling algorithms for heterogeneous processors," *8th Heterogeneous Computing Workshop (HCW' 99),* pp. 3-14, April 12, 1999.
[31] Raju D. Venkataramana and N. Ranganathan, "Multiple cost optimization for task assignment in heterogeneous computing systems using learning automata," *8th Heterogeneous Computing Workshop (HCW' 99),* pp. 137-145, April 12, 1999.
[32] Lee Wang, Howard Jay Siegel, Vwani P. Roychowdhury, and Anthony A. Maciejewski, "Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genet ic-Algorithm-Based Approach," *Journal of Parallel and Distributed Computing,* 47(1):8-22, Nov. 1997.

Biographies

Ammar Alhusaini is a Ph.D. candidate in the Department of Electrical Engineering - Systems at the University of Southern California, Los Angeles, California, USA. His main research interest is task scheduling in heterogeneous environments. Mr. Alhusaini received a B.S. degree in computer engineering from Kuwait University in 1993 and M.S. degree in computer engineering from the University of Southern California in 1996. Mr. Alhusaini is a member of IEEE, IEEE Computer Society, and ACM.

Viktor K. Prasanna (V.K. Prasanna Kumar) is a Professor in the Department of Electrical Engineering - Systems, University of Southern California, Los Angeles. He received his B.S. in Electronics Engineering from the Bangalore University and his M.S. from the School of Automation, Indian Institute of Science. He obtained his Ph.D. in Computer Science from Pennsylvania State University in 1983. His research interests include parallel computation, computer architecture, VLSI computations, and high performance computing for signal and image processing, and vision. Dr. Prasanna has published extensively and consulted for industries in the above areas. He is widely known for his pioneering work in reconfigurable architectures and for his contributions in high performance computing for signal and image processing and image understanding. He has served on the organizing committees of several international meetings in VLSI computations, parallel computation, and high performance computing. He also serves on the editorial boards of the Journal of Parallel and Distributed Computing and IEEE Transactions on Computers. He is the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is a Fellow of the IEEE.

Cauligi Raghavendra is a Senior Engineering Specialist in the Computer Science Research Department at the Aerospace Corporation. He received the Ph.D degree in Computer Science from University of California at Los Angeles in 1982. From September 1982 to December 1991 he was on the faculty of Electrical Engineering-Systems Department at University of Southern California, Los Angeles. From January 1992 to July 1997 he was the Boeing Centennial Chair Professor of Computer Engineering at the School of Electrical Engineering and Computer Science at the Washington State University in Pullman. He received the Presidential Young Investigator Award in 1985 and became an IEEE Fellow in 1997. He is a subject area editor for the Journal of Parallel and Distributed Computing, Editor-in-Chief for Special issues in a new journal called Cluster Computing, Baltzer Science Publishers, and is a program committee member for several networks related international conferences.