Run-Time Adaptation for Grid Environments *

Ammar H. Alhusaini
Computer Engineering Department
Kuwait University
P.O. Box 5969, Safat 13060 Kuwait
ammar@eng.kuniv.edu.kw

C.S. Raghavendra and Viktor K. Prasanna
Department of EE-Systems
University of Southern California
Los Angeles, CA 90089-2562
{raghu, prasanna}@usc.edu

Abstract

In this paper, we study a general mapping problem where a set of independent tasks compete for the shared resources of a Grid environment. Tasks have resource co-allocation requirements. Each task requires multiple and different resources to be allocated simultaneously. At run-time, a task may release its allocated resources during its execution and before its completion time. Our objective is to minimize the overall schedule length of all submitted tasks while satisfying all resource sharing constraints among them. We develop a two-phase mapping approach for solving this problem. The first phase of our approach is off-line planning phase where a schedule plan, which gives a scheduling order and resource assignments of tasks, is generated at compile-time. The second phase is run-time adaptation phase. The goal of the second phase is to improve the performance of the schedule plan by adapting to run-time changes such as the early release of resources and the variation in computation and communication costs. Adaptation may involve changing the scheduling order and resource assignments of the original schedule plan. Our experimental results demonstrate the effectiveness of our approach compared to a baseline algorithm that performs no adaptation at run-time and to a dynamic algorithm that performs no planning at compile-time. Our two-phase mapping approach outperforms both algorithms by up to 20% with respect to the overall schedule length.

1. Introduction

Grid Computing [8] is emerging as a new computing paradigm that exploits a wide variety of geographically distributed resources, such as supercomputers, I/O devices, storage systems, and special devices, to enable the construction of high-performance systems. Grid systems offer a consistent and inexpensive access to resources irrespective of their physical location. A Computational Grid (or simply a Grid) is also called a *Heterogeneous Computing* (HC) [13] system or a *Metacomputer* [19]. Some of the key features of grid systems are: domain autonomy, scalability, security, and interoperability between systems [8]. A major challenge in using such systems is to effectively use all the available resources. Complexities associated with the management and usage of resources across multiple administrative domains need to be hided. In a grid environment, different resources, controlled by diverse organizations with diverse policies in widely-distributed locations, need to be used together [6].

Mapping applications onto computational grids is a well-studied problem [2]. Most of the mapping algorithms in the literature are static and assume perfect estimation of computation and communication costs are available at compile-time (e.g., [1, 3, 5, 12, 17, 18, 20, 21]). However, at run-time, computation and communication costs may differ from the estimated costs and this may greatly affect the performance of static algorithms. Several dynamic algorithms have been proposed (e.g., [10, 11, 14, 15, 16]). Most static and dynamic algorithms consider compute resources only. However, it is often the case in such systems that an application requires multiple resources of different types to be allocated simultaneously. 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 links for data transfer, and a display device for interaction [9]. For such applications, co-allocation of all the required resources is necessary. In general, this problem is the *resource co-allocation* problem.

At run-time, applications may not hold all their allocated resources for the entire execution time. Some resources may be released before a task finishes execution once the task no longer needs these resources. For example, a data repository that is needed by a task to get input data may

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

be released once all input data have been retrieved. Also, scarce and expensive resources that are used at some stages of a task's execution, such as a supercomputer that is needed to process or analyze data at an early stage, can be released as soon as the task finishes using it. For these cases, run-time adaptation is needed to account for the variation in computation and communication costs and to take advantage of the early release of resources.

In this paper, we study a general mapping problem where a set of independent tasks compete for the shared resources of a computational grid. Tasks have resource co-allocation requirements where each task requires multiple and different resources to be allocated simultaneously. At run-time, a task may release some of its allocated resources during its execution and before its completion time. This early release of resources cannot be predicted at compile-time. Our objective is to minimize the overall schedule length of all submitted tasks while satisfying all resource sharing constraints among them. To solve this mapping problem, we develop a two-phase mapping approach. The first phase of our approach is *off-line planning phase* where a schedule plan is generated at compile-time. The schedule plan gives a scheduling order and resource assignments of tasks, such that the overall schedule length is minimized and all resource sharing constraints are satisfied. The second phase of our approach is *run-time adaptation phase*. The goal of this phase is to improve the performance of the schedule plan generated at compile-time by adapting to run-time changes. We develop a fast and simple window-based adaptation algorithm that accounts for run-time changes such as the early release of resources and the variation in computation and communication costs. At each mapping event, the adaptation algorithm is applied to a subset of waiting tasks based on their scheduling order in the schedule plan. Adaptation may involve advancing the execution of some tasks and changing their resource assignments specified in the original schedule plan.

Our experimental results demonstrate the effectiveness of the two-phase mapping approach. The results show the advantages of the approach compared to a baseline algorithm that performs no adaptation at run-time. Our adaptation algorithm improves the schedule length of the schedule plan generated at compile-time by up to 20%. We also compare our approach to a dynamic algorithm that performs no planning at compile-time and makes all mapping decisions at run-time. Our approach has up to 11% improvement in the overall schedule length over the dynamic algorithm. For the experiments, we vary the quality of estimated computation and communication costs in order to simulate different run-time scenarios. When the quality of estimated values is high, actual values are very close to estimated values.

The rest of this paper is organized as follows. In the next section we define our mapping problem. In Section 3, we discuss the first phase of our two-phase mapping approach: the off-line planning phase. The second phase, the run-time adaptation phase, is discussed in Section 4. Experimental results are given in Section 5. Finally, Section 6 gives the conclusions.

2. Problem Definition

2.1 System Model

We consider a computational grid with m compute resources (machines), $M = \{m_1, m_2, \ldots, m_m\}$, and a set of r non-compute resources, $R = \{r_1, r_2, \ldots, r_r\}$. Compute resources can be HPC platforms, workstations, personal computers, etc. A non-compute resource $r_k \in R$ can be a data repository, an input/output device, etc. We assume that at most one task can access any resource (compute or non-compute resource) at any given time.

System resources are interconnected by heterogeneous communication links. Estimated communication costs are given by $comm$ matrix, where $comm(r_k, m_j)$ gives the estimated cost for transferring a byte of data from resource r_k to machine m_j. $MA(m_j)$ gives a list of time slots when machine m_j is available and $RA(r_k)$ gives a list of time slots when resource r_k is available. As the mapping proceeds, these lists are updated.

2.2 Application Model

In the system under consideration, a set of n independent tasks, $T = \{t_1, t_2, \ldots, t_n\}$, compete for the shared resources of the system. We assume that the complete set of tasks to be mapped is known *a priori*. We also assume that each task t_i needs *concurrent access* to a set of resources: one compute resource and a number of additional non-compute resources as specified by its resource requirements set $R(t_i)$, where $R(t_i) \subseteq R$. The amount of data to be transferred between task t_i and resource r_k, where $r_k \in R(t_i)$, is given by $data(t_i, r_k)$. We say that task t_i and task t_j are *compatible* if $R(t_i) \cap R(t_j) = \phi$, otherwise t_i and t_j are *incompatible*. Incompatible tasks cannot be executed concurrently due to resource sharing constraints among them.

A task t_i cannot start execution until all its required resources are available. All required resources will be allocated to t_i during its execution. We assume that all required resources are acquired at the same time. These resources will be available for other tasks once released by t_i during its execution, or when it completes its execution. No information is assumed to be available at compile-time about the early release of resources.

We assume that an estimate of the computation time of a task t_i on every 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. 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

$$Exec(t_i, m_j) = ECT(t_i, m_j) + \sum_{\forall r_k \in R(t_i)} (data(t_i, r_k) \times comm(r_k, m_j))$$

where the last term gives the total time to transfer any required data between 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 of computation and communication as well as other communication models. The average execution time of task t_i, $\overline{Exec(t_i)}$, 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 estimated *start time* and *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 equal to the earliest time when m_j is available for a duration of $Exec(t_i, m_j)$. $FT(t_i, m_j)$ is defined as

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

2.3 Objective Function

Our objective is to minimize the overall schedule length (or *makespan*) of all submitted tasks while satisfying all implied resource sharing constraints among them. The objective is not to optimize the performance of an individual task. Thus, we can formally define our objective function as

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

where $Finish\,Time(t_i)$ is the completion time of task t_i and n is the total number of submitted tasks.

3. Off-Line Planning

The first phase of our two-phase approach for solving the general mapping problem defined in Section 2 is the *off-line planning* phase. The goal of this phase is to generate a valid *schedule plan* that minimizes the overall schedule length of all submitted tasks while satisfying all resource sharing constraints among them. The schedule plan gives a scheduling order and resource assignments of tasks. It also specifies estimated start and finish times of each task on all required resources. Start and finish times, as well as resource assignments, are specified by a static mapping algorithm used in this phase and are based on the estimated computation and communication costs. In this phase, we assume that all required resources will be held by a task for its entire execution time since the early release of resources cannot be predicted at compile-time.

In [4], we have developed several static algorithms for mapping applications, which are represented by Directed Acyclic Graphs (DAGs) and have resource co-allocation requirements, onto HC systems and computational grids. A *Compatibility Graph*, $g = (V, E)$, is used in [4] to capture the implied resource sharing constraints among tasks, where vertex v_i denotes task t_i and edge e_{ij} exists if and only if t_i and t_j are incompatible. The idea of the algorithms is based on selecting maximal sets of independent tasks and execute them concurrently. An *independent set* of an undirected graph is defined as a set of vertices such that no two vertices of the set are adjacent [7]. An independent set is called a *maximal independent set* if there is no other independent set that contains it [7]. A maximal independent set of the compatibility graph, g, represents a maximal set of tasks that have no precedence and resource sharing constraints among them and can be executed concurrently. Since independent tasks representation can be considered as a special case of DAG representation, the idea of the algorithms developed in [4] can be used in the off-line planning phase to generate a schedule plan. Figure 1 shows the pseudo code of our off-line planning algorithm. The algorithm is based on the Highest Average-Execution-Time First (HAETF) algorithm developed in [4]. Additional details about HAETF algorithm can be found in [4].

4. Run-Time Adaptation

The second phase of our two-phase mapping approach is the *run-time adaptation* phase. The goal of this phase is to improve the performance of the schedule plan generated in the first phase by adapting to run-time changes. At run-time, actual computation and communication costs may differ from the estimated costs used in the off-line planning phase to generate the schedule plan. Also, tasks may release some of their allocated resources during their executions and before their finish times (as opposed to the assumption made in the first phase). Therefore, we need to adapt to run-time changes and we may need to modify the original schedule plan (i.e., the scheduling order and resource assignments of tasks) in order to improve the overall scheduling length of all submitted tasks.

We develop a fast and simple window-based algorithm for run-time adaptation. The scheduling order and resource

Off-Line Planning Algorithm

Begin
Initialize:
  T={ all submitted tasks }, $ALLOCATED = \phi$, and $Sch\_Plan = \phi$.
  $length = 0$.
Construct the compatibility graph g.
Pick a task t_c with the highest $\overline{Exec(t_c)}$ from T.
Find a maximal independent set of tasks S from T such that $t_c \in S$.
While T is not empty do:
  Sort tasks in a non-increasing order of their $\overline{Exec(t_i)}$.
  For each task t_i in S (in their order) do:
    Assign a compute resource m_j to t_i in order to minimize its finish time $FT(t_i, m_j)$.
    Add t_i (with its start time and finish time, and resource assignments) to Sch_Plan.
    Update $MA(m_j)$ and $RA(r_k), \forall r_k \in R(t_i)$.
    If ($FT(t_i, m_j) > length$) then $length$=$FT(t_i, m_j)$.
  Add all tasks in S to $ALLOCATED$ and remove them from T.
  Let t_x be the allocated task with the lowest finish time $FT(t_x, m_j)$.
  Remove t_x from $ALLOCATED$.
  Let $C = T$, where C is the set of candidate tasks that can be allocated next.
  Remove all tasks from C that are incompatible with any allocated task.
  If ($C \neq \phi$):
    Pick a task t_c with the highest $\overline{Exec(t_c)}$ from C.
    Find a maximal independent set of tasks S from C such that $t_c \in S$.
End (while)
End

Figure 1. Pseudo code of the off-line planning algorithm

assignments of the original schedule plan are used in our adaptation algorithm. The algorithm starts by executing the first maximal independent set of tasks as ordered in the schedule plan. Then, the algorithm proceeds by considering a subset of waiting tasks at each mapping event in order to adapt to run-time changes. Adaptation may involve changing the scheduling order and resource assignments of the selected subset of tasks.

A mapping event is defined as the time when our run-time adaptation is applied to a selected set of tasks. Mapping events can be repeated at fixed time intervals (e.g., every 100 seconds), every time a task finishes execution, or every time a resource becomes available. In general, more frequent mapping events leads to a better adaptation. On the other hand, the cost of adaptation increases as the frequency of mapping events increases. The frequency of mapping events can be modified and adjusted dynamically at run-time. In our approach we choose mapping events to occur every time a resource becomes available.

The reason of only selecting a subset of tasks (a window of tasks) to be considered at each mapping event is to minimize the cost of adaptation. Considering all waiting tasks will be not only expensive but also adversely affects the original schedule plan. We want to avoid thrashing and ensure that the performance of our adaptation is no worse than no-adaptation approach.

Pseudo code of our run-time adaptation algorithm is shown in Figure 2. The algorithm starts by executing the first maximal independent set of tasks as selected by the off-line planning algorithm (Steps 3-4). The start time of these tasks in the original schedule plan is equal to 0. These tasks will be executed on their assigned machines in the schedule plan. Our adaptation algorithm cannot be applied to this set of tasks since there is no information available yet about any variation in computation and communication costs or early release of resources. Then the adaptation algorithm proceeds as follows until all tasks are executed. A subset of tasks, S, is selected at each mapping event starting from the first waiting task based on the scheduling order of the schedule plan (Step 8). The size of S is equal to a pre-selected value of *window-size*. In Step 9, all tasks that cannot be executed at this mapping event are removed from S. Then, all tasks in S are considered for execution one-by-one in their scheduling order in the schedule plan. For each task t_i we first find the machine m_b that gives the best finish time for t_i at this mapping event. Then, we compare m_b to machine m_j that has been assigned to t_i by the off-line planning algorithm as specified in the schedule plan. Based on this comparison, we decide if we would execute t_i on machine m_b at this mapping event or not. The comparison can be one of the following cases:

1. $m_b = m_j$
   Machine m_b is the same machine selected by the off-line planning algorithm for task t_i and most likely it is the best suited machine for t_i. Therefore, our approach for this case is to execute t_i on m_b at this mapping event.

2. $m_b \neq m_j$
   Machine m_b is different than the machine selected by the off-line planning algorithm for task t_i. For this case, we compare the performance of both machines based on the estimated execution time of t_i on each machine as follows:

   (a) $Exec(t_i, m_b) = Exec(t_i, m_j)$
   Our approach for this case is to execute task t_i on machine m_b at this mapping event because m_b has the same performance as m_j.

   (b) $Exec(t_i, m_b) < Exec(t_i, m_j)$
   In this case, it is clear that machine m_b is better suited for task t_i than m_j but the off-line algorithm did not assign it to t_i. We have a good opportunity at this time to improve the finish time of t_i by executing it on m_b. Therefore, our approach is to execute t_i on m_b at this mapping event.

   (c) $Exec(t_i, m_b) > Exec(t_i, m_j)$
   For this case, our experiments show that it is not always a good idea to execute t_i on m_b especially if $Exec(t_i, m_b) \gg Exec(t_i, m_j)$. Even though m_b is expected to be the best machine for t_i at this time, but the schedule length will suffer if the actual execution time of t_i on m_b is much greater than $Exec(t_i, m_b)$. A small difference between the actual and the estimated value of the execution time can have a large effect on the schedule length if $Exec(t_i, m_b) \gg Exec(t_i, m_j)$. Our approach for this case is to execute t_i on m_b at this mapping event if $Exec(t_i, m_b) \leq (Exec(t_i, m_j) + \Delta\ Exec(t_i, m_j))$, where Δ (Delta) is a value between 0% and 100%.

The complexity of our run-time adaptation algorithm is a function of number of tasks, number of mapping events, and window size.

5. Experimental Results

We conducted extensive simulations to evaluate the performance of our two-phase mapping approach. To define a grid system, number of machines (*no_machines*) and number of resources (*no_resources*) are given to a software simulator as inputs. Estimated communication costs among all resources are selected randomly from a uniform distribution with a mean equal to *ave_comm*.

The workload consists of independent tasks that are randomly generated as follows. The total number of tasks,

Run-Time Adaptation Algorithm

Begin

Let sch_plan be the schedule plan generated in the off-line planning phase.
$counter$ = 0.
Run the first maximal independent set of tasks as ordered in sch_plan:
    using the scheduling order and resource assignments of sch_plan. Update $counter$ after each execution.
$counter$ points now to the first task that cannot run at this time (i.e., time 0).
While ($counter <$ total number of tasks) do:
    At each mapping event do:
        Select a subset of tasks, S, with a size equals to *window-size* ,starting from $sch\_plan[counter]$.
        Remove all tasks that cannot run at this time from S (preserve the scheduling order).
        For each task t_i in S (in their scheduling order) do:
            Find the best machine m_b to execute t_i at this time.
            Execute t_i on m_b if $Exec(t_i, m_b) \leq (Exec(t_i, m_j) + \Delta\ Exec(t_i, m_j))$,
                where m_j is the machine assigned to t_i in sch_plan.
            $counter$++.
End(while)

End

Figure 2. Run-time adaptation algorithm

no_tasks, and average computation cost of a task, *ave_comp*, are given as inputs. The estimated computation cost of each task on every machine is randomly selected from a uniform distribution with a mean equal to *ave_comp*. Resource requirements for each task are randomly selected from available resources. Number of required resources is randomly selected from a uniform distribution with a mean equal to $\sqrt{no\_resources}$. 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 actual (run-time) values of computation and communication costs are randomly selected from a uniform distribution within the range [-PE,+PE] of the estimated values, where PE (Percentage Error) is a value between 0% and 100%. Perfectly estimated values correspond to PE=0.

For our experiments, *no_machines* was set to 10 and *no_resources* was set to 20. Tasks were generated with *ave_comp*=50 and *ave_data_size*=300 Kbyte. PE was set to 50%, Δ was set to 15%, and window size (*window-size*) was set to 25% of total number of tasks. We use two performance metrics to evaluate the performance of our mapping approach. These metrics are: (1) the overall schedule length, and (2) adaptation cost (time) on a Sun Enterprise 250 with 1 Gbyte of memory. We compare our approach to a baseline algorithm that performs no adaptation at run-time. The baseline algorithm is called No-Adapt algorithm. Each data point in the figures of this section is an average of 25 distinct runs.

Figure 3 compares our approach and No-Adapt algorithm with different numbers of tasks ranging from 100 tasks to 500 tasks with increments of 100 tasks. Our approach has up to 20% improvement in schedule length over No-Adapt algorithm. Figure 3 clearly shows the advantage of our approach over No-adapt algorithm that performs no adaptation at run-time. As shown in Figure 4, the cost of our adaptation algorithm is very small compared to expected execution times of applications in HC systems and computational grids. The cost of adaptation is less than 1 second with 500 tasks.

The comparison of our approach and No-Adapt algorithm with different *window-size* values is shown in Figure 5 and Figure 6. The total number of tasks in both figures is 300 tasks. We changed the value of *window-size* from 15 to 120 tasks with increments of 15 tasks. This corresponds to changing *window-size* values from 5% to 40% of total number of tasks with increments of 5%. Figure 5 shows that the improvement in schedule length of our approach over No-Adapt algorithm increases as the window size increases. On the other hand, the cost of our adaptation algorithm also increases as the window size increases as shown in Figure 6. This is because, as window size increases, more tasks are examined by our algorithm at each mapping event. For this experiment, we found that the performance of our approach is relatively the same with *window-size*$\geq$120 tasks (40% of total number of tasks). This is due resource sharing constraints, which prevent tasks to run at the same time. Thus, increasing the window size would not help finding more tasks that can run concurrently if resource sharing among tasks is high.

The effect of PE value on the performance of our approach is shown in Figure 7 for a set of 300 tasks. We changed the value of PE from 25% to 100% with increments of 25%. As shown in the figure, the improvement in schedule length of our approach over No-Adapt algorithm increases as PE value increases. This clearly shows that the advantage of our run-time adaptation approach increases as the quality of estimated computation and communication costs decreases. No-Adapt algorithm, which uses estimated values only and does not adapt to run-time changes, is greatly affected by the quality of estimated costs.

Figure 8 shows the effect of Δ values (defined in Section 4) on the performance of our mapping approach for a set of 300 tasks. In the figure, we changed the value of Δ from 10% to 90% with increments of 10%. As shown in the figure, the improvement in schedule length of our approach over No-Adapt algorithm increases as Δ value increases until 25%. After that, the performance of our approach decreases. Recall that Δ is used in our run-time adaptation algorithm to compare the performance of two machines in order to make a decision regarding advancing the execution and changing the machine assignment of a task t_i. If the value of Δ is high (depends on PE value), a small difference between actual and estimated computation and communication costs can have a large effect on the schedule length (as explained in Section 4). In our experiments, we found that adaptation may lead to worse results than No-Adapt algorithm when the total number of tasks is small and the difference between estimated and actual computation and communication costs is small (i.e., the quality of estimated values is high).

In order to demonstrate the effectiveness of our off-line planning algorithm, we compare our approach to a dynamic algorithm that performs no planning at compile-time. Initially, the algorithm, called No-Plan algorithm, randomly orders all tasks. Then, it examines a subset, S, of tasks for mapping at each mapping event. The size of S is equal to *window-size*. A task, which can be executed at a mapping event, is mapped to its best machine at that time. Figure 9 shows the comparison between our approach and No-Plan algorithm with respect to schedule lengths. The frequency of mapping events and the window size are the same in both approaches. As shown in Figure 9, our approach has up to 11% improvement in schedule length over No-Plan algorithm. This clearly shows the advantage of our off-line planning algorithm.

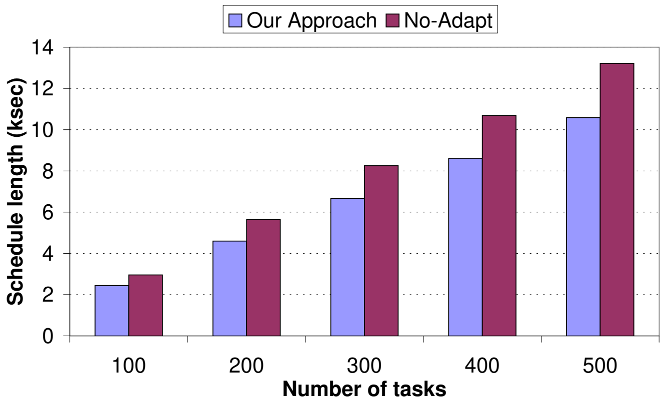

Figure 3. Comparison of schedule lengths with different number of tasks

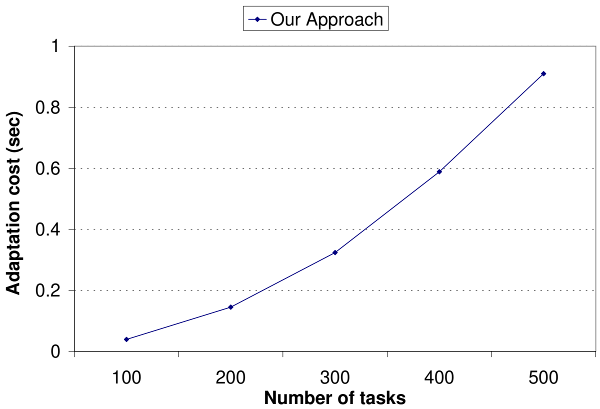

Figure 4. Adaptation cost with different number of tasks

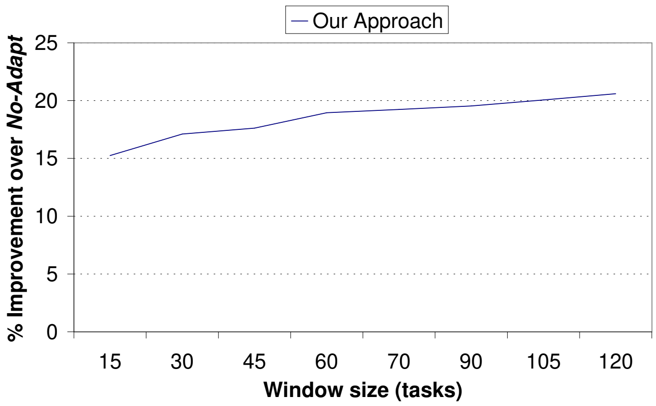

Figure 5. Improvement (in schedule length) of our approach over No-Adapt algorithm with different window sizes

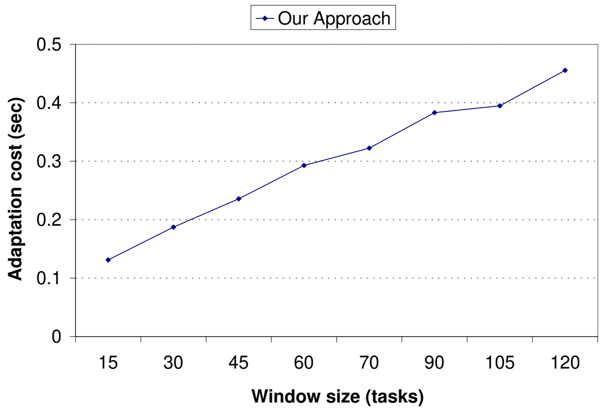

Figure 6. Adaptation cost with different window sizes

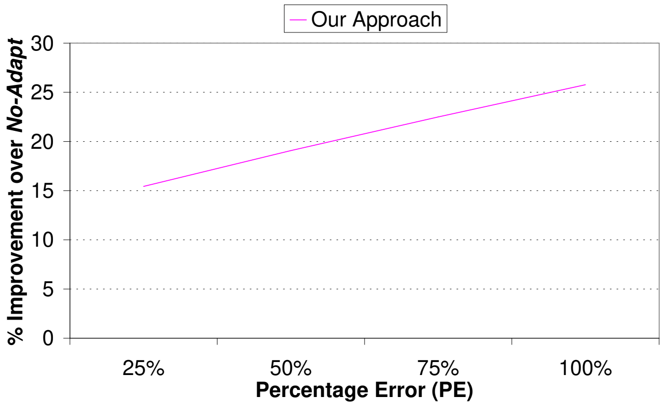

Figure 7. Improvement (in schedule length) of our approach over No-Adapt algorithm with different PE values

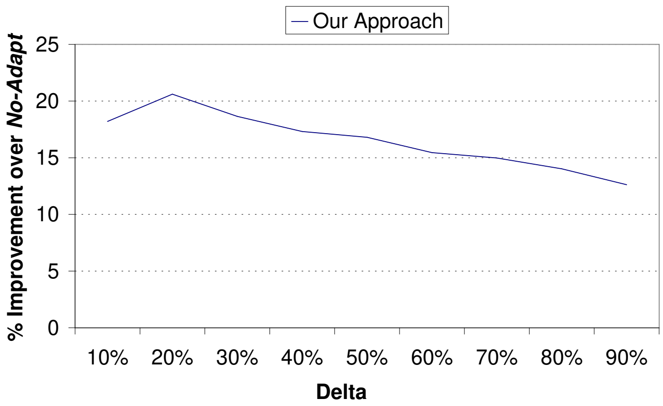

Figure 8. Improvement (in schedule length) of our approach over No-Adapt algorithm with different Δ values

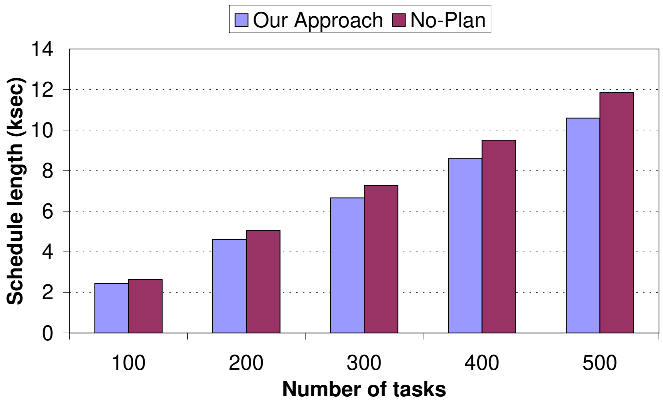

Figure 9. Comparison with No-Plan algorithm

6. Conclusions

In this paper, we have developed a two-phase approach for mapping a set of independent tasks onto a grid environment. Tasks have resource co-allocation requirements where each task requires multiple and different resources to be allocated simultaneously. The first phase of our approach is the off-line planning phase where a schedule plan is generated at compile-time with the objective of minimizing the overall schedule length of all submitted tasks while satisfying all resource sharing constraints among them. The second phase is the run-time adaptation phase where run-time changes, such as the variation in computation and communication costs and the early release of resources, are considered in order to improve the performance of the schedule plan generated at compile-time.

We have evaluated the performance of our two-phase mapping approach with simulation results. The experimental results showed the advantages of our run-time adaptation algorithm compared to a baseline algorithm that performs no adaptation at run-time. Our adaptation algorithm improved the schedule length of the schedule plan generated at compile-time by up to 20%. The results also showed that the our approach has up to 11% improvement in the schedule length over a dynamic algorithm that performs no planning at compile-time. This showed the advantage of our off-line planning algorithm. In general, the experimental results demonstrated the effectiveness of our two-phase mapping approach.

References

[1] I. Ahmad and Y.-K. Kwok. On parallelizing the multiprocessor scheduling problem. *IEEE Trans. on Parallel and Distributed Systems*, 10(4):414–432, April 1999.

[2] A. H. Alhusaini. *A unified mapping framework for heterogeneous computing systems and computational grids*. PhD thesis, University of Southern California, 2001.

[3] A. H. Alhusaini, V. K. Prasanna, and C. S. Raghavendra. A unified resource scheduling framework for heterogeneous computing environments. In *8th Heterogeneous Computing Workshop (HCW' 99)*, pages 156–165, April 1999.

[4] A. H. Alhusaini, V. K. Prasanna, and C. S. Raghavendra. A framework for mapping with resource co-allocation in heterogeneous computing systems. In *9th Heterogeneous Computing Workshop (HCW' 2000)*, pages 273–286, May 2000.

[5] T. Braun, H. J. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, and R. Freund. A comparison study of static mapping heuristics for a class of meta-task on heterogeneous computing systems. In *8th Heterogeneous Computing Workshop (HCW' 99)*, pages 15–29, April 1999.

[6] R. Buyya, S. Chapin, and D. DiNucci. Architectural models for resource management in the Grid. In *1st IEEE/ACM International Workshop on Grid Computing*, pages 18–35, Bangalore, India, December 2000.

[7] N. Christofides. *Graph theory: An algorithmic approach*. Academic Press, 1975.

[8] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a new computing infrastructure*. Morgan Kaufmann, San Francisco, CA, 1999.

[9] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that support advance reservations and co-allocation. In *Intl. Workshop on Quality of Service*, 1999.

[10] R. Freund, B. Carter, D. Watson, E. Keith, and F. Mirabile. Generational scheduling for heterogeneous computing systems. In *Intl. Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA '96)*, pages 769–778, August 1996.

[11] M. Iverson and F. Ozguner. Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment. In *7th Heterogeneous Computing Workshop (HCW' 98)*, pages 70–78, March 1998.

[12] M. Iverson, F. Ozguner, and G. J. Follen. Parallelizing existing applications in a distributed heterogeneous environment. In *4th Heterogeneous Computing Workshop (HCW' 95)*, pages 93–100, April 1995.

[13] A. Khokhar, V. K. Prasanna, M. Shaaban, and C. L. Wang. Heterogeneous computing: challenges and opportunities. *IEEE Computer*, 26(6):18–27, June 1993.

[14] C. Leangsuksun, J. Potter, and S. Scott. Dynamic task mapping algorithms for a distributed heterogeneous computing environment. In *4th Heterogeneous Computing Workshop (HCW' 95)*, pages 30–34, April 1995.

[15] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *8th Heterogeneous Computing Workshop (HCW '99)*, pages 30–44, April 1999.

[16] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *7th Heterogeneous Computing Workshop (HCW '98)*, pages 57–69, March 1998.

[17] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund. Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environment. In *5th Heterogeneous Computing Workshop (HCW' 96)*, pages 98–117, April 1996.

[18] 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.

[19] L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):45–52, June 1994.

[20] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *8th Heterogeneous Computing Workshop (HCW' 99)*, pages 3–14, April 1999.

[21] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):8–22, November 1997.

Biographies

Ammar Alhusaini is an assistant professor in the Computer Engineering Department at Kuwait University. He received a B.S. degree in computer engineering from Kuwait University and M.S. and Ph.D. degrees in computer engineering from the University of Southern California. His main research interest is resource management and task mapping in grids environments and heterogeneous computing systems. He is a member of IEEE, IEEE Computer Society, and ACM.

Cauligi Raghavendra is a research professor in the Department of Electrical Engineering-Systems at the University of Southern California. 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, Kluwer Publishers, and is a program committee member for several networks related international conferences.

Viktor K. Prasanna (V. K. Prasanna Kumar) received his BS in Electronics Engineering from the Bangalore University and his MS from the School of Automation, Indian Institute of Science. He obtained his Ph.D. in Computer Science from the Pennsylvania State University in 1983. Currently, he is a Professor in the Department of Electrical Engineering as well as in the Department of Computer Science at the University of Southern California, Los Angeles. He is also an associate member of the Center for Applied Mathematical Sciences(CAMS) at USC. He served as the Division Director for the Computer Engineering Division during 1994-98. His research interests include parallel and distributed 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 has served on the organizing committees of several international meetings in VLSI computations, parallel computation, and high performance computing. He is the Steering Co-chair of the International Parallel and Distributed Processing Symposium [merged IEEE International Parallel Processing Symposium(IPPS) and the Symposium on Parallel and Distributed Processing(SPDP)] and is the Steering Chair of the International Conference on High Performance Computing(HiPC). He serves on the editorial boards of the Journal of Parallel and Distributed Computing, IEEE Transactions on Computers, and the IEEE Transactions on Parallel and Distributed Systems. He was the founding Chair of the IEEE Computer Society Technical Committee on Parallel Processing (TCPP). He is a Fellow of the IEEE.