

Chapter 1. Introduction

Different kinds of resources interconnected with a high-speed network provide a computing platform, called *Heterogeneous Computing* (HC) system [11]. In general, a HC system means the system that only consider the compute resources which can be a workstation, a personal computer, etc. But, there are many kinds of other resources in HC system, high performance computing platform, workstations, personal computers, data repository, input/output device, etc. There are some applications in HC system. 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 [16]. Because of the various and sufficient resources, HC system can support a powerful execution capability. Therefore, an efficient and effective mapping algorithm for an application on HC system becomes more important. A good scheduling method will enormously promote the execution capability of HC system.

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 resource (matching problem) and ordering task execution in time (scheduling problem) to optimize a specific object function. Many static [1, 2, 3, 4] and dynamic [5, 6, 7, 8, 9] algorithms are proposed for mapping applications in HC system (for a detail classification see [10]). Most of the previous algorithms focus on compute resources only.

In this thesis, we consider the problem of mapping a set of applications to 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. In this research area, Alhusaini is a pioneer [11]. He proposed two methods for resource

co-allocation problem [11, 12]. In [11] and [12], both of them are two phases algorithms. These two algorithms almost are the same except the second phase. In the first phase, a schedule plan is generated at compiler 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 goal of the second phase is to improve the performance of the schedule plan generated at compiler time by adapting to run time change. In a communication intensive application, Alhusaini's method will suffer a disadvantage, that is, schedule length increases quickly caused by the communication cost.

In order to overcome the disadvantage in Alhusaini's method, we also proposed a two phases algorithm which is called the *dynamic resource co-allocation algorithm*. In the first phase, we will only generate the data that will be used in the second phase. The main allocation mechanism is in the second phase. We successfully overcome the disadvantage in Alhusaini's method and propose an effective and efficient algorithm for resource co-allocation problem.

The thesis is organized as follows. In chapter 2, we will describe the problem domain, system model, application model, and some terminologies. The algorithm which we propose will be introduced in chapter 3. In chapter 4, we will describe our simulation environment and evaluate our algorithm. Finally, we will make the conclusion and list some future work in chapter 5.

Chapter 2. Fundamental Background and Related Work

In this chapter, we will introduce the system model, application model, and some basic terminologies in section 2.1. Next, we will introduce the resource co-allocation problem in the section 2.2. Finally, we will go through some related works to get familiar with the development of the research in this area in section 2.3.

2.1 Fundamental Background

2.1.1 System Model

In our system model, we consider a heterogeneous computing system with m compute resources (machines), $M=\{m_1, m_2, \dots, m_m\}$, and a set of r non-compute resources (resource), $R=\{r_1, r_2, \dots, r_r\}$. A machine can be a HPC platform, a workstation, a personal computer, etc. A non-compute resource can be a data repository, an input/output device, etc. Resources are interconnected by the network.

2.1.2 Application Model

Before describing our application model, we will define the DAG which is often used to represent the parallel program. We assume that a parallel program is composed of n tasks $\{T_1, T_2, \dots, T_n\}$ in which there is a partial order. The partial order $T_i < T_j$ implies that T_j can not start execution until T_i finishes due to the data dependency between them. This restriction is also called the precedence constrain. Formally, we give the following definition and Figure 2.1 shows a simple example of DAG.

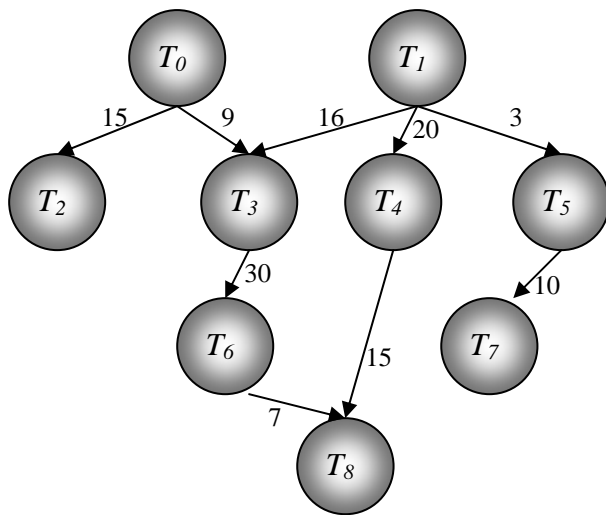


Figure 2.1 An example of DAG

Exec =

6	10	12	19
30	7	35	30
20	18	54	13
4	15	20	5
19	6	22	26
15	3	12	20
9	7	1	6
26	30	26	14
7	1	10	4

Figure 2.2 computation cost matrix

Definition 2.1 A parallel program can be represented by a *Directed Acyclic Graph* (DAG) $G = (T, E, C, R)$, where

- $T = \{T_1, T_2, \dots, T_n\}$ is a finite set of tasks;
- E is a set of edges which is between the tasks in T and each edge $e(i, j)$ represents the data dependency between task T_i and task T_j ;
- C is the function from E to integer in which c_{ij} represents the communication cost from task T_i to task T_j .
- R is a function from T to a set of resources in our system. When a task T_i is executed, it needs a set of resources, denoted as $R(T_i)$.

We assume that the communication cost in the same machine is negligible in our system. That is, the communication cost is zero if T_i and T_j are scheduled to the same machine.

In a DAG, a task without any parent is called an *entry task* and a task without any

child is called *exit task*. If there exists the data dependency from task T_i to task T_j in the DAG, we say that task T_i is the *immediate predecessor* of task T_j and task T_j is the *immediate successor* of task T_i .

In our system model, a task on different machine has different computation cost. Thus, we need the computation cost matrix $Exec$ to describe the computation cost. The matrix $Exec$ is defined as below.

Definition 2.2 In a given task graph, a computation cost matrix $Exec$ is a $n \times m$ matrix in which each component $Exec(T_i, m_j)$ is the computation cost to complete T_i on machine m_j . We assumed that the communication cost between machine m_j and all resources which task T_i needs to access during execution is included in the $Exec(T_i, m_j)$.

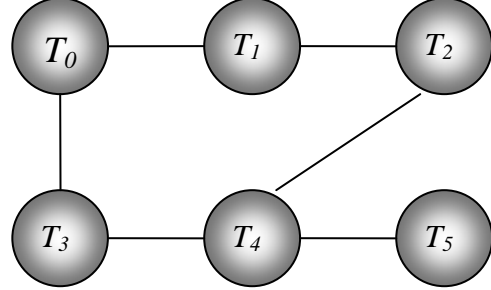


For example, suppose there are four machines in our system. Figure 2.2 shows the corresponding computation cost matrix of the DAG in Figure 2.1. A submitted application consists of several tasks and is modeled by DAG. In our heterogeneous computing system, we consider a set of applications, $A = \{A_1, A_2, \dots, A_n\}$, compete for system resources. We assumed that the whole set of applications to be mapped is known apriori (static applications).

2.1.3 Basic Terminologies

In this section, we will define some terminologies. For a set of machines, $M = \{m_1, m_2, \dots, m_m\}$, and a set of non-compute resources, $R = \{r_1, r_2, \dots, r_r\}$, $MA(m_j)$ and $RA(r_k)$ gives the earliest available time of machine m_j and resource r_k , respectively.

Task	Resource Requirements
T_0	r_1, r_2
T_1	r_2, r_3
T_2	r_3, r_5
T_3	r_1, r_4
T_4	r_4, r_5, r_6
T_5	r_6



(a)

(b)

Figure 2.3 Compatible graph example (a) A set of tasks and their resource requirements (b) Compatible graph

As the mapping proceeds, the earliest available time of a resource is defined as the release time of the last task assigned to it.



Definition 2.3 In a given partial schedule, we define the *Earliest Start Time* of task T_i on machine m_j , denoted as $EST(T_i, m_j)$, by the following formula:

$$EST(T_i, m_j) = \max\{MA(m_j), \max\{FT(T_k) + C_{k,i}\}\} \quad (\text{Formula 2.1})$$

where $FT(T_k)$ is the finish time of task T_k and $C_{k,i}$ is the communication cost between task T_k and T_i .

Definition 2.4 In a given partial schedule, we define the *Earliest Finish time* of task T_i on machine m_j , denoted as $EFT(T_i, m_j)$, by the following formula:

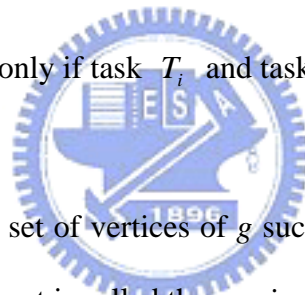
$$EFT(T_i, m_j) = EST(T_i, m_j) + Exec(T_i, m_j) \quad (\text{Formula 2.2})$$

Definition 2.5 Two tasks T_i and 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 constraints among them. This is the resource sharing constraint. Therefore, tasks may be unable to run concurrently for precedence constraint or resource sharing constraint. We use the *compatibility graph* defined below to capture the implied resource sharing constraints among tasks that may belong to the same or different application. For example, Figure 2.3(b) shows the corresponding compatibility graph of the tasks set in Figure 2.3(a).

Definition 2.6 Given a set of tasks, $T = \{T_1, T_2, \dots, T_n\}$, and their resource requirements, the *compatibility graph* [11] $g = (V, E)$, where

- V is the set of tasks T ;
- Edge $e(i, j)$ exists if and only if task T_i and task T_j are incompatible.



An *independent set* is a set of vertices of g such that no two vertices of the set are adjacent. An independent set is called the *maximal independent set* if there are no other independent set of g that contains it. A maximal independent set with the largest number of vertices among all maximal independent set is called a *maximum independent set*. The maximum independent set problem is NP-complete [15]. In our model, a maximal independent set of g represent a maximal set of tasks that can be executed concurrently if there is no precedence constraint among them. For example, in Figure 2.3 (b), the maximal independent sets of g are $\{T_0, T_4\}$, $\{T_1, T_4\}$, $\{T_0, T_2, T_5\}$, $\{T_1, T_3, T_5\}$, and $\{T_2, T_3, T_5\}$. The last three sets are maximum independent sets. Since the maximum independent problem is NP-complete, we will use a heuristic approach to select maximal independent sets. The approach is based on first selection a critical task v_c , and then finding a maximal independent sets that contains v_c .

2.2 The Resource Co-Allocation Problem

The resource co-allocation problem can be defined as the problem of simultaneously allocating multiple resources of different type to applications in order to meet specific performance. The need of resource co-allocation is a common characteristic of application running in Heterogeneous Computing (HC) environment. For example, an application may require a data repository, a High Performance Computing (HPC) platform, multiple display devices, and communication links all to be allocated simultaneously.

These are assumptions in our model. We assumed that only one task can use any resource (compute and non-compute resource) at any given time. For the resources requirement of each task, we assumed that each task T_i needs concurrent access to one compute resource m_j and a number of additional resources as specified by the set $R(T_i) \in R$. A task T_i cannot start execution until all its required resources are available to it. All required resources will be allocated to the task during its execution, and may be released before its completion. For the estimated execution time of each task, we assume that the estimated execution time and the actual execution time (run time) are different, that is, we can not perfect predict the execution time of each task in compiler time.

The objective function in our framework is to determine an assignment of tasks to compute resources and schedule their executions bases on all machines such that the overall schedule length of all submitted applications is minimized. Thus, we can define our objective function as

$$\text{Minimize}\{\max_{i=1}^N[\text{FinishTime}(A_i)]\}, \quad (\text{Formula 2.3})$$

where $\text{Finish Time}(A_i)$ is the completion time of application A_i .

2.3 Related Work

In this section, we will briefly introduce two related work based on our problem domain in previous section. There are some differences between them. The method in section 2.3.1, it doesn't consider early resource releasing but another one does.

2.3.1 Alhusaini's Method without Early Resource Release [11]

The method that proposed by Alhusaini is a two phases algorithm. The first phase is compiler-time mapping phase and the other is run-time adaptation phase. We will introduce their concept as follows.

The main work in compiler time mapping phase is to generate a schedule plan (scheduling order and resource assignment of tasks). In the compiler-time mapping phase, given a set of applications and resource requirements of tasks, Alhusaini's method first find tasks that have satisfied precedence constrains. For simplicity, Alhusaini's method combined all submitted applications by a hypothetical zero-cost entry task. Each entry node of submitted applications will connect to the hypothetical zero-cost entry task and the weight of these edges are zero. Alhusaini's method proceeds level-by-level as follows. For each level l of G , it constructs the compatibility graph g for all tasks in this level. Then it repeats two steps until all tasks in this level have been allocated. The first process is to find a maximal independent set s . The second process is to allocate all tasks in s according to the scheduling order of the tasks. The mechanisms of deciding the scheduling order of a maximal independent set and different strategies for selecting critical tasks are listed in [11]. The reason of selecting critical tasks is mentioned in section 2.1.3.

The run-time adaptation phase is used when the actual execution time is different from the estimated execution time. The goal of this phase is to dynamically adjust the scheduling order to get a better performance. In compiler-time mapping phase, it will generate a order list. Once a task completes its execution, this phase will scan through the order list to find all tasks that can be executed at this time and make local reordering.

2.3.2 Alhusaini's method with Early Resource Release [12]

Alhusaini [12] proposed another algorithm by releasing a resource r_k if the task that holds the resource r_k in run time won't use it again. Thus, these two algorithms almost are the same except for the run-time adaptation phase.

The run-adaptation phase is used while a mapping event is happened, where a mapping event can be repeated at fixed time intervals, every time a task finishes, or every time a resource become available. Each process of mapping event is processed as follows. A sub set of tasks, S , that can be executed now is selected starting from the first waiting task based in the scheduling order of the scheduling plan that produce at compiler-time mapping phase. All tasks in S are considered for execution one-by-one in their scheduling order in the schedule plan. For each task T_i , it will first find the best machine m_b that gives the shortest finish time for T_i at this mapping event. Then it uses a comparison condition to decide a machine to task T_i . Based on this comparison, it decides if we would execute T_i on machine m_b or m_j that has been assigned to T_i in compiler time mapping phase as specified in schedule plan at this mapping event. The comparison condition (migration condition) is

$$Exec(T_i, m_b) \leq Exec(T_i, m_j) + \Delta Exec(T_i, m_j), \quad (\text{Formula 2.4})$$

where Δ is a value between 0% and 100%. It will execute T_i on machine m_b if the condition is true. In the next chapter, we will describe the disadvantages in Alhusaini's method and how we overcome these advantages.

We extend this method to a set of tasks not only for a set of independent tasks. In chapter 4, we will use this method to be the baseline algorithm. Then, we will evaluate the improvement of our algorithm compared to this baseline algorithm.



Chapter 3. Dynamic Resource Co-allocation Algorithm

In this chapter, we will mention the motivation of our method and define some terminologies that used in our algorithm in section 3.1. In section 3.2, we will describe our algorithm. Finally, we will have a discussion in section 3.3.

3.1 Motivation of our Method

To begin with, we know that Alhusaini's method with early resource releasing will allocate a machine $m_j \in M$ in our system to each task T_i in DAG at compiler-time. For each task T_i , it will find a best machine m_b which has the earliest finish time among all machines. Next, it uses a migration condition to decide whether the task T_i need to be migrated to machine m_b or not while machine m_b and machine m_j are different. This may result in one situation that task T_i will not be allocated to machine m_b because the migration condition doesn't satisfy. We can easily find that when the heterogeneity is getting larger, the migration condition is more difficult to be satisfied. Furthermore, if task T_i doesn't be allocated to machine m_b , the finish time of task T_i will not be the earliest. This is the first disadvantage of Alhusaini's migration mechanism. For example, Figure 3.1 (b) is the estimated execution time of task T_2 on each machine. We assume that m_j and m_b are m_2 and m_1 , respectively. Testing the migration condition by the estimated execution times of task T_2 on machines m_2 and m_1 , we can find that the migration condition will be false.

In the migration condition, we can find that the migration condition doesn't consider the communication cost between task T_i and its immediate predecessors. This

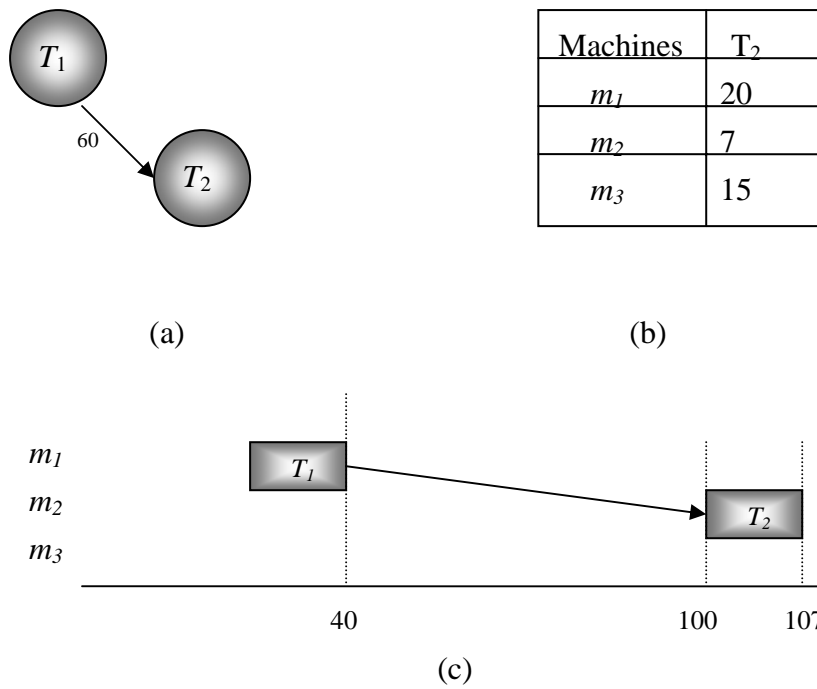


Figure 3.1 (a) part of DAG (b) estimated execution time of task T_2 on each machine (C) part of schedule plan

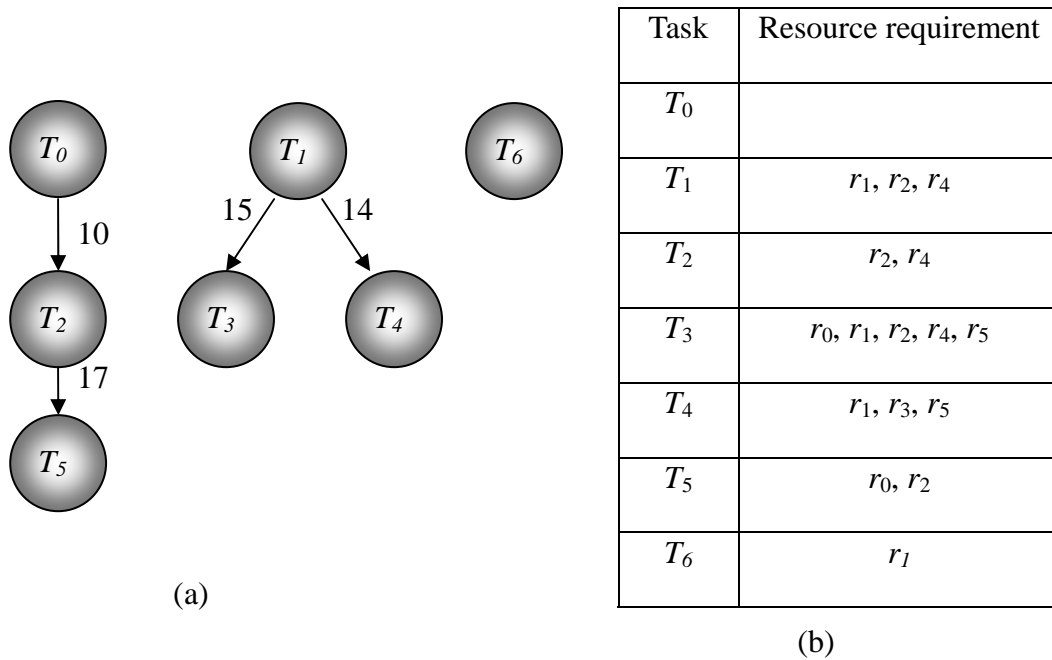
will result in the situation that the schedule length increases quickly caused by the communication cost especially for the communication intensive applications. For example, in Figure 3.1, we assume that there are three machines and the Task T_1 is allocated to machine m_1 in the run time. Figure 3.1(a) is part of the DAG for a communication intensive application and Figure 3.1(b) is the estimated execution time of task T_2 on each machine. Moreover, we assume that the task T_2 is allocated to machine m_2 in the compiler time phase. We will consider how to allocate the task T_2 in run-time adaptation phase now. First, it will find a best machine which is m_1 now because the estimated earliest finish time of task T_2 on machine m_1 is the shortest that is calculated by Formula 2.2. Then, when we test the migration condition, we find that the migration condition is false even if the value, Δ , is 100%. Therefore in Figure 3.1(c), the task T_2 will be allocated to machine m_2 and completes at time unit 107. We

can find that if task T_2 was allocated to machine m_1 , the finish time of task T_2 will be 60. In a communication intensive application, the schedule length will increase quickly caused by migration condition fail. This is another impact of migration condition in communication intensive application. Because the migration condition mechanism has this disadvantage, we would try to remove it in our algorithm.

In Alhusaini's method, the major part of the algorithm focuses on the allocation mechanism in compiler-time mapping phase. But we have no any information about when a task T_i will release a resource $r_k \in R(T_i)$ before task T_i completes in compiler time. And, we don't know what resources will be available at any significant time unit. So, all tasks are assumed that they won't release any resource before they complete in compiler time. In order to improve the performance, Alhusaini use an adaptation mechanism in run time. Here, we want to allocate a set of independent tasks at each mapping event in run time directly instead of the compiler time. And, we will dynamic select a set of tasks to be allocated depending on the number of common resources and the data dependency between tasks which are unscheduled. Before we describe how we modify Alhusaini's method, we will describe the weight function and the *weighted compatibility graph* in our method.

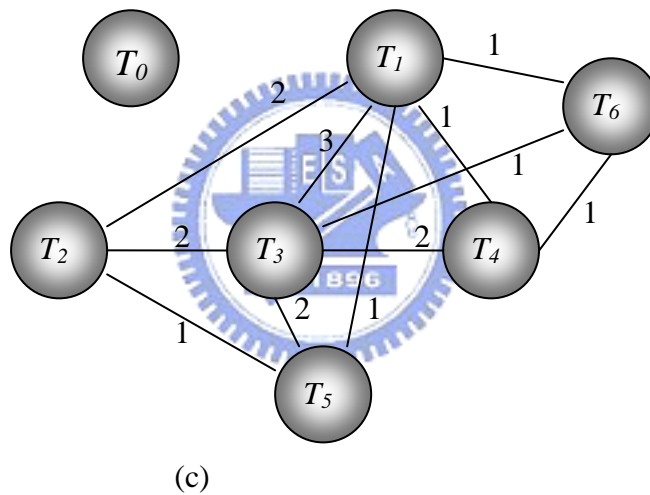
3.2 Dynamic Resource Co-allocation Algorithm

We will describe the basic principle of *Dynamic Resource Co-allocation Algorithm* (DRCA) for resource co-allocation problem in this section. DRCA also has two phases: one is called the compiler time phase and the other is run time allocation phase. In section 3.2.1, we will describe the compiler time phase first. Next, we will describe the run time allocation phase in section 3.2.2.



(a)

(b)



(c)

Figure 3.2 (a) two DAGs (b) resource requirement of tasks in Figure 3.2 (a) (c) weighted compatibility graph

3.2.1 Compiler Time Phase

In the compiler time phase, we only produce the data that will be used in run time allocation phase. There are only two steps in this phase. Step one is to construct the *Weighted Compatible Graph* (WCG). The definition of WCG is given below.

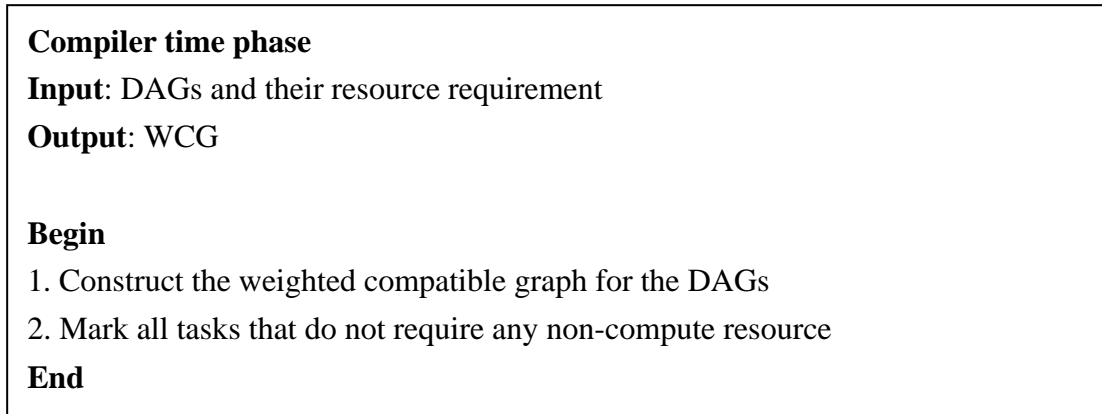
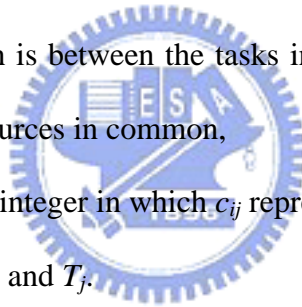


Figure 3.3 Pseudo code of the compiler time phase

Definition 3.1 Given a set of tasks and the resource requirement of each tasks, the *Weighted Compatibility Graph (WCG)* $W = (T, E, C)$, where

- T is the set of all tasks
- E is a set of edges which is between the tasks in T ; edge $e(i, j)$ exists if task T_i and task T_j use some resources in common,
- C is a function form E to integer in which c_{ij} represents the number of common resources between task T_i and T_j .



For example, Figure 3.2 (c) shows the corresponding weighted compatibility graph to the set of tasks in Figure 2.3 (a) and their resource requirements in Figure 3.2 (b). The second step is to mark all tasks that do not require any non-compute resource during execution. Figure 3.3 is the algorithm of compiler time phase.

3.2.2 Run Time Allocation Phase

To begin with, a task T_i is in the ready state if all the immediate predecessors of task T_i are completed and every resource $r_k \in R(T_i)$ are free and each unscheduled task

Run time allocation phase

Begin

1. Let WCG be the weighted compatible graph generated in the compiler time phase
2. Counter = 0
3. P is an integer such that $2^P < N$, where N is the total number of tasks
4. **While** (counter < total number of tasks) **do**:
5. At each mapping event do:
6. If (some non-compute resources are released in this mapping event)
7. Adjust WCG
8. Put all ready state tasks into the set *READY*
9. Extract all tasks that are marked at compiler time from *READY* and put them into set *M*
10. **do**
11. Pick one task t with the highest weight in *READY* to be critical task and put it into set *C*
12. Remove tasks that are incompatible to t in *READY*
13. Calculate the size z of *READY*
14. **While** ($z > P$)
15. Find a maximal independent set S from *READY* such that $C \subseteq S$
16. Reinsert all tasks from *M* into S
17. **While** (S is not empty) **do**:
18. **If** (C is not empty) **do**:
19. Pick the highest weight task t from C
20. Remove t from C
21. Remove t from S
22. **Else do**:
23. Pick a task t from S
24. Remove t from S
25. Find a machine m_j that has earliest finish time to t
26. Allocate $R(t)$ to t
27. Execute t on machine m
28. Counter++
29. **End(while)**
30. **End(while)**
31. **End**

Figure 3.4 Pseudo code of run time allocation phase

T_i in our method has a dynamic weight. The weight of each task is defined as follows.

Definition 3.2 Given a DAG $G = (T, E, C)$ and the WCG $W = (T_1, E_1, C_1)$ of this DAG, we define the weight $w(i)$ for each task T_i as

$$w(i) = out_degree(i) + \sum_{e(i, j) \in E_1} c_{ij}, \quad (\text{Formula 3.1})$$

where $out_degree(i)$ is out degree of task T_i in DAG and c_{ij} is the number of common resources between task T_i and T_j in WCG.

For example, given the DAG in Figure 3.2 (a) and the WCG in Figure 3.2 (c), the weight of task T_1 , $w(1)$, is ten. And $w(3)$ and $w(4)$ are ten and four, respectively. During run time, if task T_i releases a resource r_k and r_k is one of the resources required by an unscheduled task T_j , the value c_{ij} will minus one. Therefore, the value c_{ij} will adjust in run time as the mapping proceeds. And, the weight of task T_i , $w(T_i)$, will adjust in run time as the mapping proceeds. For example, while task T_1 is in the execution state and releases resource $r_1 \in R(T_1)$, the weights of task T_3 , T_4 , and T_6 will become nine, three, and two respectively.

One of the steps in run time allocation phase is to select a maximal independent set S . But the time complexity of selecting a maximal independent set is really great. The time complexity is $O(2^C CR)$, where C is the number of tasks in the independent set not included the critical task v_c , and R is the total number of non-compute resources. Therefore, in order not to increase the overhead in run time, we have to add some processes to restrict the integer C . There are two kinds of situations that we describe below.

First, if a task T_i doesn't require any non-compute resource and is in the ready state, it must be selected into the maximal independent set S . The reason is that task T_i

will not suffer any resource sharing constrain between other tasks that is in ready state, too. Therefore, it can concurrently execute with other tasks that is also in the ready state. We can mark these tasks that do not require any non-compute resource in the compiler time and extract them before selecting the maximal independent set S . Then, we will reinsert them to the maximal independent set S after selecting the maximal independent set.

Second, for the selecting of maximal independent set, we select a set of critical tasks such that the integer C is less than k , where k is the number such that 2^k is less than the total number of tasks. While we select more critical tasks, there are more non-compute resources will be reserve for them. So, there are more tasks that in the ready state suffer resource sharing constrain to critical tasks. We can remove them before selecting a maximal independent set. This process will have the time complexity of selecting maximal independent set to be $O(NR)$, where N is the total number of tasks.

In run time allocation phase, there are some other processes at each mapping event. To ensure precedence constrain to be satisfied, we gather all tasks that are in the ready state to be set *READY*. Before we select a maximal independent set from *READY*, we must extract all tasks that is marked in compiler time phase. This will be helpful for us to restrict the time complexity of selecting a maximal set from *READY*. As we mention above, we will reinsert them to the maximal independent set.

We will select a set of tasks to be critical tasks according to the weight of each task. A task that has the greatest weight in the set *READY* will be selected first. There are two reasons for a task T_i to have the greatest weight in *READY*. First, task T_i has many immediate successors or second, there are many tasks that are unscheduled and

	T_0	T_1	T_2	T_3	T_4	T_5	T_6
M_1	5	5	4	7	8	3	3
M_2	7	6	4	6	7	8	4
M_3	4	4	5	5	6	5	3

(a)



(b)



(c)

Figure 3.5 (a) the estimated computation cost of tasks in Figure 3.2 (a) (b) the result of first mapping event (c) the result after second mapping event at time unit 2

require the resources that need by task T_i , too. Therefore, if the largest weight task T_i completes earlier, we will have more tasks to be ready and we may select more tasks to be execute concurrently. Selecting a set of critical tasks will also help us reduce the time complexity of selecting maximal independent set.

After deciding the critical tasks, we will select a maximal independent set S that included the critical tasks from *READY* to be allocated. For the allocation mechanism, we use the highest weight first for the critical tasks in S and the other tasks are random. We use a simple allocation mechanism and consider two situations to reduce the time complexity of selecting maximal independent set in our algorithm. This will help us reduce the overhead in run time. If the overhead is increase quickly in run

time, the schedule length will also increase. When we want to allocate a task T_i to a suitable machine, we use the earliest finish time of task T_i on each machine to decide which machine is suitable. Figure 3.4 is the pseudo code of the run time allocation phase.

In Figure 3.5, we show an example for the first mapping event of tasks in Figure 3.2 (a) and Figure 3.5 (a) is their estimated execution time on each machine. There are seven tasks in the Figure 3.2 (a), so the value k in run time allocation algorithm is 1. There two tasks will be put in the set *READY* that is task T_0 and task T_1 . Because task T_0 doesn't require any non-compute resource, it will be extracted before the selecting of maximal independent set. The result of first mapping event is shown in Figure 3.5 (b). We assumed that task T_1 will release resource r_1 at time unit 2. The result after this mapping event is in Figure 3.5 (c).



3.3 Discussion

In section 3.1, we ever mentioned that Alhusaini's method has two disadvantages caused by the migration condition. We will describe how we avoid these disadvantages in our method. For the first disadvantage in Alhusaini's method, that is, in a communication intensive application, the schedule length will increase quickly caused by communication cost. To avoid this disadvantage, we must consider the computation and communication cost simultaneously. Therefore, we use the earliest finish time of a task T_i on each machine to choose a machine m_j that will make the task T_i completes earlier and allocate task T_i to machine m_j . This will help us avoid the schedule length increasing quickly in a communication intensive application. In chapter4, we will illustrate the effect of migration condition through the simulation

	Compiler time	Run time
Alhusaini's method	$O(2^N NR)$	$O(N^2 R)$
DRCA	$O(N^2 R)$	$O(N^2 R)$

Figure 3.6 Time complexity of Alhusaini's method and DRCA

result.

In Alhusaini's method, when the heterogeneity is getting bigger and bigger, the migration condition would be false frequently. It would be harder to allocate a task T_i to the best machine m_b . In DRCA, we directly allocate a task T_i to a machine m_j in run time. Therefore, we can avoid the disadvantage of migration condition mechanism.

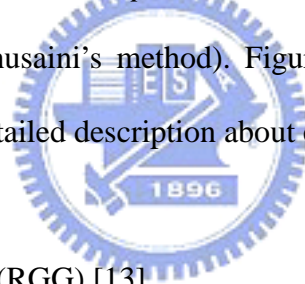
Figure 3.6 is the time complexity of Alhusaini's method and DRCA. In the compiler time mapping phase of Alhusaini's method, the time complexity of selecting a maximal independent set is $O(2^N NR)$, where N is the total number of tasks and R is the total number of resources. The run time adaptation phase of Alhusaini's method, the time complexity is $O(N^2 R)$. In DRCA, the time complexity of compiler time phase is focus on constructing the WCG. The time complexity of run time allocation phase in DRCA is $O(N^2 R)$.

Chapter 4. Simulation and Performance Evaluation

After describing the Dynamic Resource Co-allocation Algorithm (DRCA), we will verify the effectiveness of it by implementation and simulation. In section 4.1, we will describe the architecture of simulator. Next, we will evaluate the performance in section 4.2.

4.1 Simulation Construction

We use the C++ language to construct our simulator. There are two parts in our simulator. The first part is *Random Graph Generator (RGG)* [13], the second part is the algorithm (DRCA or Alhusaini's method). Figure 4.1 is the flow chart of our simulator. We will give the detailed description about each part in the following.



(a) Random Graph Generator (RGG) [13]

The main function of the RGG is to generate the DAG. As we define in definition 2.1, the parallel program with n tasks can be represented as DAG with n tasks. In our simulation, we use several parameters to generate a DAG. The parameters are as following.

- Number of tasks in the graph, ($TASK$).
- Shape parameter of the graph, ($SHAPE$).

The height (depth) of a DAG is randomly generated from a uniform distribution

with a mean value equal to $\frac{\sqrt{TASK}}{SHAPE}$. The wide of each level is randomly selected

from a uniform distribution with mean equal to $SHAPE \times \sqrt{task}$. A dense graph

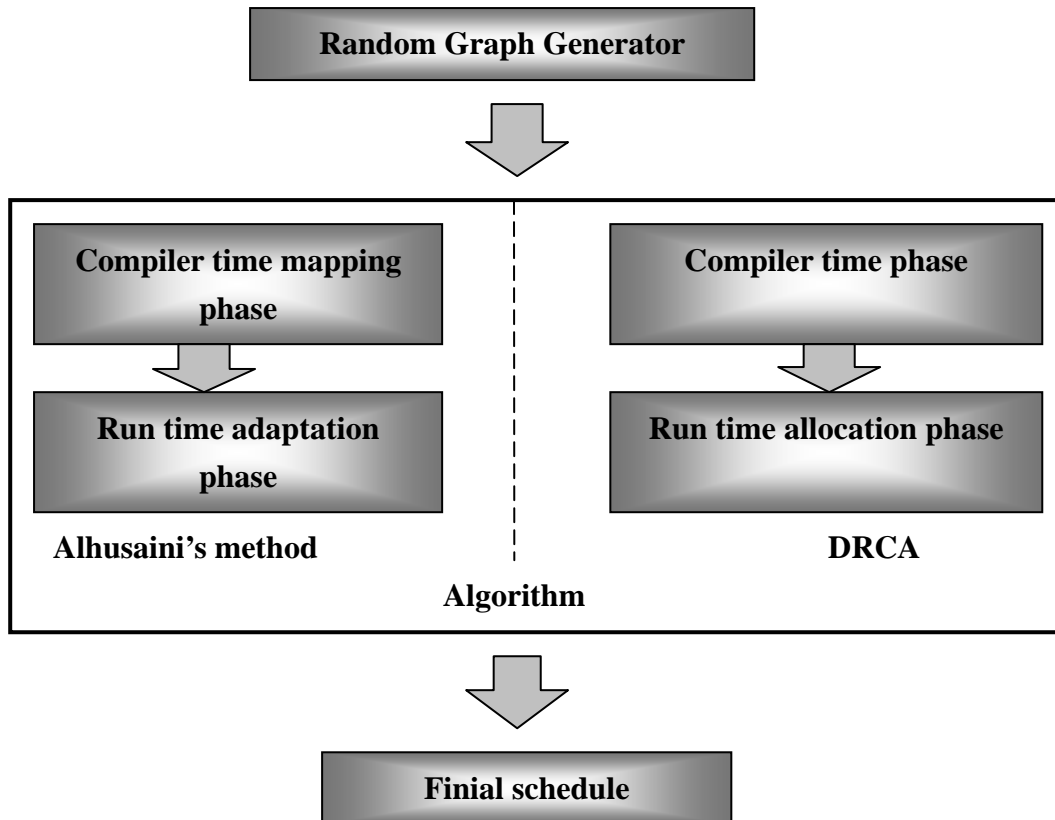


Figure 4.1 The flow chart of the simulator

(a shorten graph with high parallelism) can be generated by selecting $SHAPE \gg 1.0$; if $SHAPE \ll 1.0$, it will generate a long graph with a low parallelism degree.

- Maximal out degree of a task (OUT_DEGREE).
- Communication to computation ratio (CCR).

It is the ratio of the average communication cost to the average computation cost. If the CCR value of a DAG is very low, it can be considered as a computation-intensive application. On the contrary, it will be a communication-intensive application.

- Range percentage of computation cost on machines, ($HETEROGENEITY$).

It is basically the heterogeneity factor for machine speeds. A high percentage

value causes a significant different in a task's computation cost among the machines and a low percentage indicates that the expected computation cost of a task is almost equal on any given machines in the system.

The parameters that we describe above can be used to generate DAGs with any characteristic. In the following, we list other parameters used in our simulation environment.

- Number of machines in the system, (*MACHINE*)
- Number of non-compute resources in the system, (*RCS*)
- Percentage Error, (*PE*).

The actual (run-time) values of computation and communication costs are randomly selected from the range $[-PE, +PE]$ of the estimated values, where *PE* is a value between 0% and 100%. Perfect estimated values correspond to $PE = 0$.

- *RANGE*.

It is the ratio of the number of tasks that doesn't have perfect estimated values to the total number of tasks.

- Resource Using Mode, (*RUM*).

This mode is used to decide that how many resource a task uses. There are three modes in RUM. *RANDOM_RUM*: random decides that how many resources a task uses. *MORE_RUM*: almost all tasks use more than half of the total number of non-compute resources. *LESS_RUM*: almost all tasks use less than half of the total number of non-compute resource.

- Resource Releasing Mode, (*RRM*)

This mode is used to decide that when a task T_i will release a resource $r_k \in R(T_i)$

before completes. There are four modes in RRM. RANDOM_RRM: random decide that when a task T_i will release a resource $r_k \in R(T_i)$. LATE_RRM: almost all tasks release a task $r_k \in R(T_i)$ after half of the actual computation cost. EARLY_RRM: almost all tasks release a task $r_k \in R(T_i)$ before half of the actual computation cost. NO_RRM: no early resources release.

Using all of these parameters, we can construct any kinds of applications and characteristic to them. In each experiment, we run ten DAGs to each set of parameters and use the average schedule lengths of these DAGs to be the schedule length of this set of parameters. Figure 4.2 shows all range of the parameters.

(b) Algorithm

Alhusaini's method is a level-by-level algorithm. In order to get a shorter schedule length in Alhusaini's method, we implement the Alhusaini's method in a crossing level method [14] in our simulator. DRCA is also implemented in the simulator. The input of the algorithm is the DAGs and their resource requirements. The output of the algorithm is the final schedule. In the next section, we will illustrate our evaluation results.

4.2 Performance Evaluations

In this section, we will evaluate the performance of DRCA comparing with the Alhusaini's method. First of all, we define the *Schedule Length Ratio* (SLR) as the average schedule length of DRCA divides by the average schedule length of Alhusaini's method. If the SLR is larger than 1.0 means the Alhusaini's method has

MACHINE	2, 4, 6, 8, 10, 12
PE	0%, 30%, 60%, 90%
RANGE	20%, 40%, 60%, 80%, 100%
OUT_DEGREE	4, 8, 12
SHAPE	0.5, 1, 2
RCS	4, 8, 12, 16, 20
RRM	RANDOM, LATE, EARLY, NO
RUM	RANDOM, MORE, LESS
CCR	0.1, 0.5, 1, 5, 10
TASK	100, 150, 200, 250, 300
HETEROGNEEITY	0.2, 0.5, 1, 1.5

Figure 4.2 The range of all parameters

the smaller schedule length. On the contrary, if the SLR is smaller than 1.0 means the DRCA has smaller schedule length. We observe the variation of SLR on three parameters, CCR, HETEROGENEITY, and TASK in our experiments.

First, we will observe the effect of CCR. The simulation result is illustrates in Figure 4.3 and Figure 4.4. We fixed some parameters in these experiments, MACHINE{8}, PE{30%}, RANGE{60%}, OUT_DEGREE{4, 8, 12}, SHAPE{0.5, 1, 2}, RCS{8}, RANDOM_RUM, and RANDOM_RRM. We observe the effect of CCR {0.1, 0.5, 1, 5, 10} with changing the TASK {100, 150, 200} and HETEROGENEITY{0.2, 1, 1.5} in Figure 4.3 and Figure 4.4, respectively. We can find that as the CCR increases the SLR decreases quickly. Therefore, in a communication intensive application, Alhusaini's method will have larger schedule length than DRCA. In a communication intensive application, an efficient method

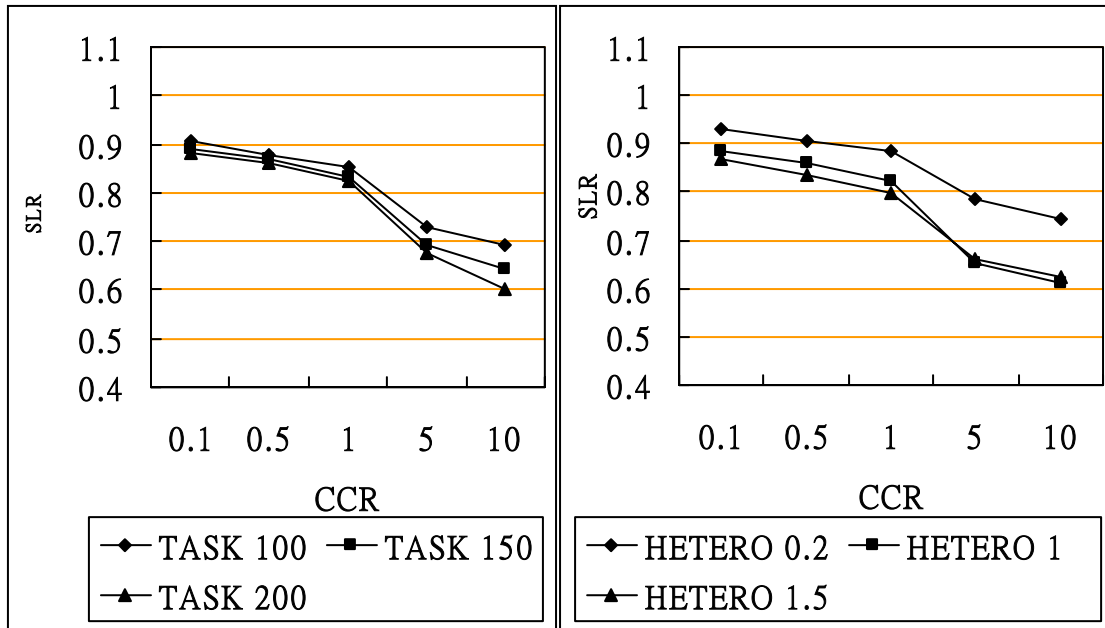


Figure 4.3 The simulation result of effect of CCR with changing TASK

Figure 4.4 The simulation result of effect of CCR with changing HETEROGENEITY

to reduce the communication cost between tasks will enormously shorten the schedule length. But, the migration condition in Alhusaini's method, only computation cost will be used to decide to migrate a task to its best machine or not in run time. Therefore, the schedule length will increase quickly caused by the communication cost. As we mention in section 3.1, Alhusaini's method has this disadvantage and we prove it through this experiment. In a computation intensive application, we do not have the great improvement like communication intensive application but the SLR is also below to 1.0. So, we still have shorter schedule length in a computation intensive application.

There is also another disadvantage in Alhusaini's method that we mention in section 3.1, that is, as the HETEROGENEITY is getting larger the migration condition will be false frequently. In Figure 4.5 and Figure 4.6, we will illustrate the effect of HETEROGENEITY on SLR. Through these two experiments, we can find

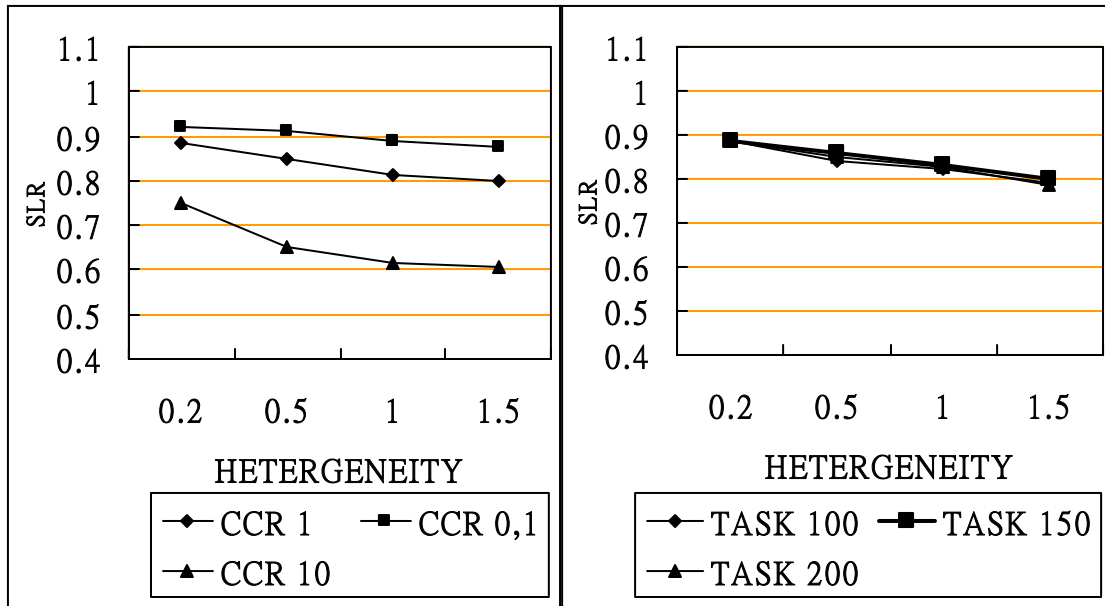


Figure 4.5 The simulation result of effect of HETEROGENEITY with changing CCR

Figure 4.6 The simulation result of effect of HETEROGENEITY with changing TASK

that the SLR decreases as the HETEROGENEITY increases. Each time, the migration condition is tested and the result is false. Here, we consider the task T_i to be allocated to the machine that has larger finish time than best machine. This will make the finish time of the task T_i not the earliest. Therefore, if the migration condition false frequently, there are more tasks won't be allocated to the best machine. While the HETEROGENEITY becomes larger, the difference of computation cost of a task on each machine will be greatly. So, if the machine m_j that has been allocated to a task T_i in compiler time has smaller computation cost than other machines in system for task T_i , it will be harder to migrate successfully in run time. This is because the computation cost of task T_i on other machines is bigger than the computation cost on m_j . Therefore, as the HETEROGENEITY increases the SLR decreases.

We also want to observe the parameter TASK. When there are more tasks in a DAG, there are more and more chances for a DAG to test the migration condition and

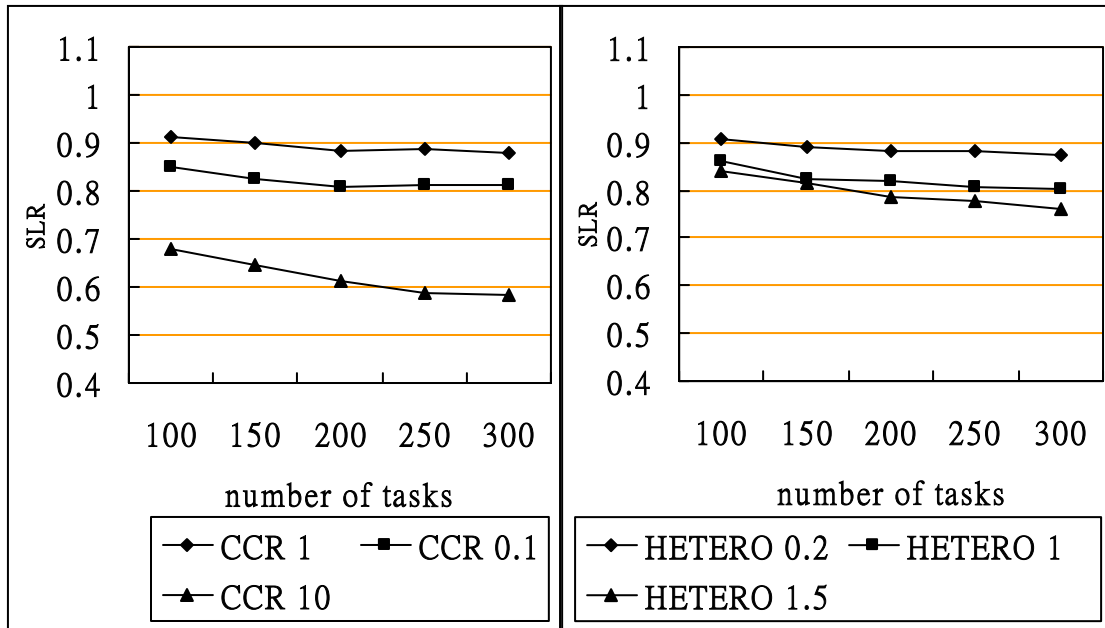


Figure 4.7 The simulation result of effect of TASK with changing CCR

Figure 4.8 The simulation result of effect of TASK with changing HETEROGENEITY

to be false frequently. While there are more times of migration condition false, there are more times for Alhusaini's method to increase the schedule length. Therefore, as the number of tasks increases the SLR decreases. In Figure 4.7 and Figure 4.8, we observe the parameter TASK. We find that the SLR decreases slowly as the TASK increases.

Because of the disadvantage of migration condition in Alhusaini's method, we get much improvement in DRCA. But beside the disadvantage of migration condition, we want to know whether we have any other improvement in DRCA comparing with Alhusaini's method. Therefore, in order to make sure that each task will be allocated to best machine, we set the value, Δ , in Alhusaini's method to be infinity. Therefore, each task will be allocated to best machine at run time adaptation phase in Alhusaini's method. The simulation results are in Figure 4.9 to Figure 4.14. In these experiments, we still observe three parameters, CCR, HETEROGENEITY, and TASK. Through

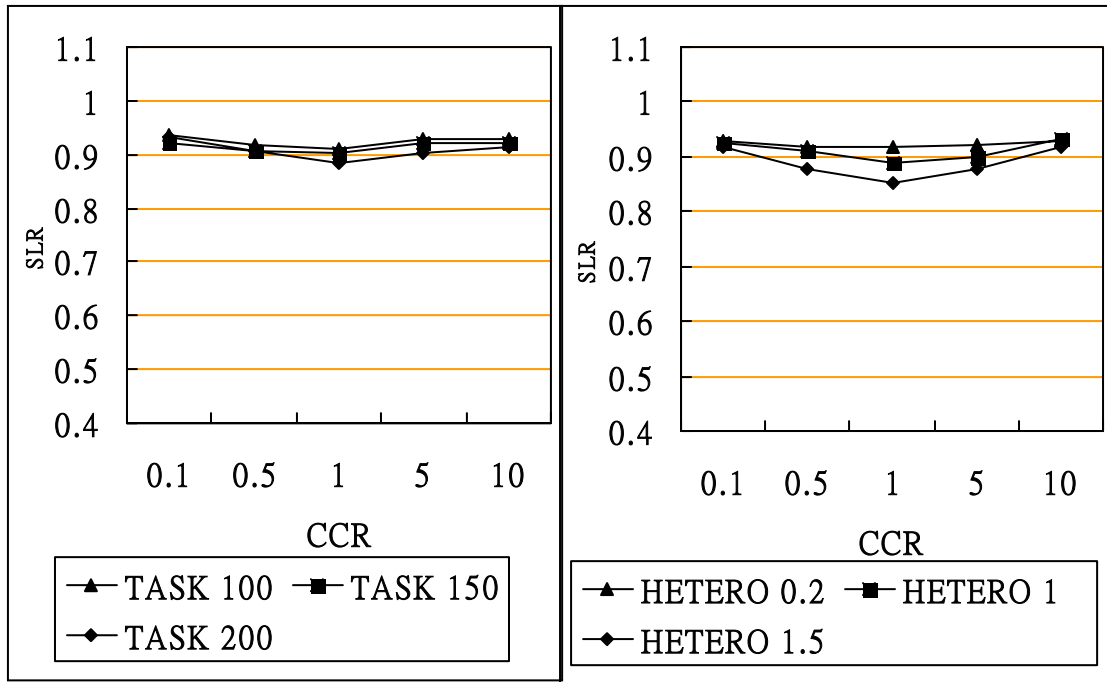


Figure 4.9 The simulation result of effect of CCR without effect of migration condition

Figure 4.10 The simulation result of effect of CCR without effect of migration condition

Figure 4.9 to Figure 4.14, we can find that the average SLR is about 0.9. This result shows that without effect of migration condition, DRCA still have shorter schedule length than Alhusaini's method.

In our simulation, we found that the parameters, RCS, MACHINE, PE, RANGE, RUM, and RRM have no any significant impact on the SLR. In the simulation results of these parameters, the varying range of them is like the range in Figure 4.2. We found that as these parameters change in the simulation the SLR almost has no variation. Therefore, we do not discuss the simulation results of these parameters here.

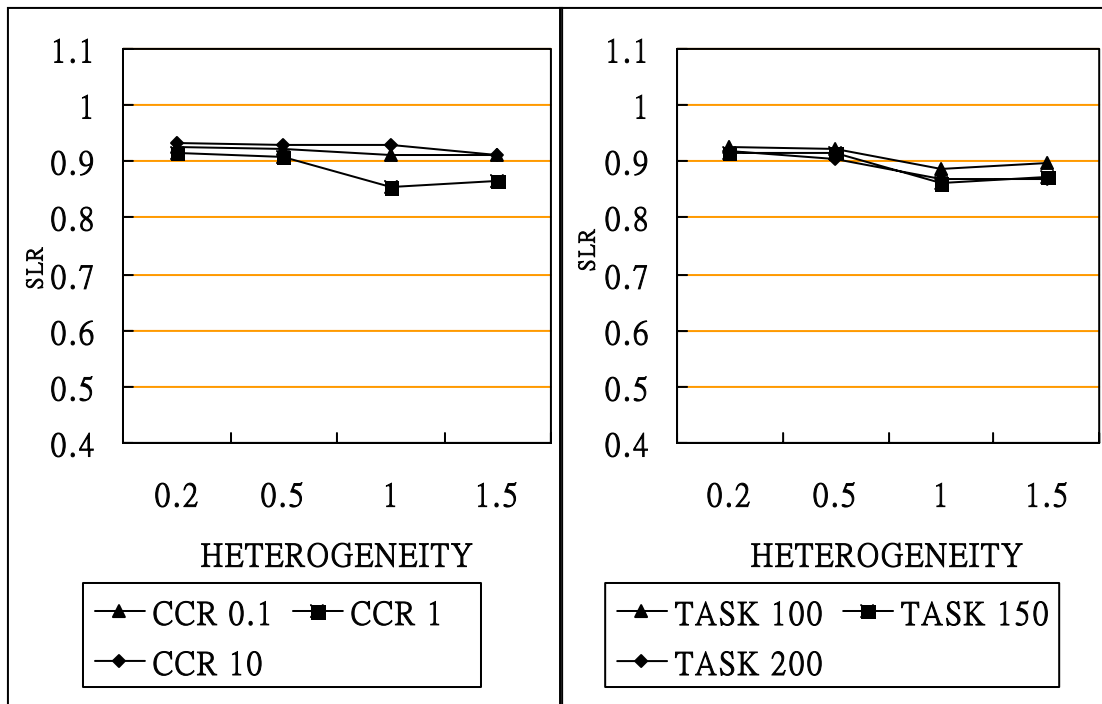


Figure 4.11 The simulation result of effect of HETEROGENEITY without effect of migration condition

Figure 4.12 The simulation result of effect of HETEROGENEITY without effect of migration condition

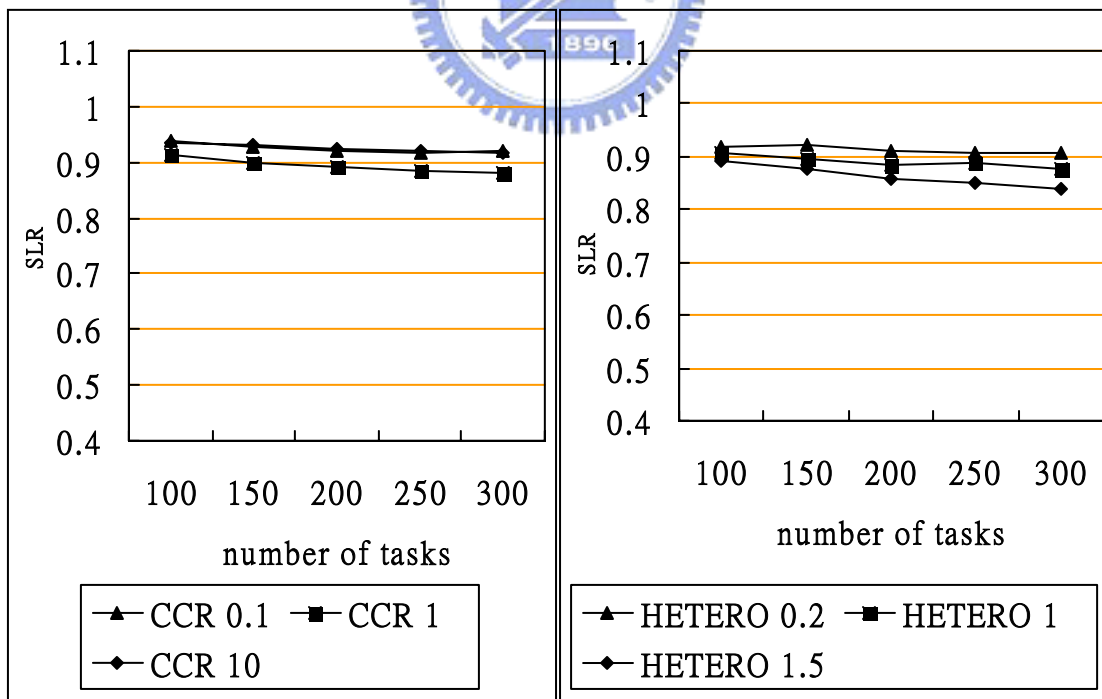



Figure 4.13 The simulation result of effect of TASK without effect of migration condition

Figure 4.14 The simulation result of effect of TASK without effect of migration condition

Chapter 5. Conclusion and Future Work

In the previous chapters, we have introduced our system model, application models, and proposed a two phase algorithm for the resource co-allocation problem. The compiler time phase is used to generate the data that will be used in run time. The run time allocation phase is used to select a maximal independent set and allocate a task to a suitable machine. In order to evaluate the performance of our algorithm, we construct a simulation environment and compare with Alhusaini's method. Finally, we will conclude our thesis and propose some future work for our research.

5.1 Conclusion



In Alhusaini's method, we found that the migration condition mechanism would be a drawback for finding a better schedule plan. And, the assumption in compiler time doesn't consider early resource releasing. For these reasons, we proposed the dynamic resource co-allocation algorithm. In summary, it has the following main advantages compared with Alhusaini's method:

- (1) For effectiveness, we propose a weight function for allocating a task in run time and remove the effect of migration condition mechanism. We verify the effectiveness of DRCA by constructing simulation. The simulation results in chapter 4 show that DRCA effectively shortens the schedule length comparing with Alhusaini's method. Especially in communication intensive application, we obtain more better performance.
- (2) For efficiency, the time complexity of run time phase in Alhusaini's method and DRCA are in common. Both of them are $O(N^2R)$, where N is the total number of

tasks and R is the total number of non-compute resources. But in compiler time phase, the time complexity of Alhusaini's method is $O(2^N R)$ greater than DRCA's time complexity which is $O(N^2 R)$. Therefore, whether in compiler time or run time, we are more efficient than that of Alhusaini's method.

5.2 Future Works

In addition to the feature we discussed before, there are still some issues in the future research.

- (1) In our system, we use a fully-connected network. However, for more realistic, we can take the consideration of different network topology, system latency..., etc, in our system model. When the system model is more and more realistic, it is more difficult to design a good algorithm.
- (2) We may expand our assumptions to consider usage of multiple compute resources and advance resource reservations. With advantage reservation, system resources can be reserved in advance for specific time intervals. 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 required duration.

Bibliographies

- [1] 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.
- [2] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund, "Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environment," 5th *Heterogeneous Computing Workshop (HCW'98)*, pp. 98-117, Apr 1996.
- [3] G. C. Sih and E. A. Lee, "A compiler-time scheduling heuristic for interconnection-constrained heterogeneous processor architecture," *IEEE Trans. On Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175-187, Feb. 1993.
- [4] L. Wang, H. J. Siegel, V. Roychowdhury, and A. Maciejewski, "Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm –Based Approach," *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, pp.8-22, Nov. 1997.
- [5] I. Ahmad and Y. Kwok, "On parallelizing the multiprocessor scheduling problem," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 4, pp.414-432, April 1999.
- [6] 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 (PDPTA '96)*, pp. 769-778, Aug. 1996.
- [7] M. Iverson, and F. Ozguner, "Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment," 7th *Heterogeneous Computing Workshop (HCW'98)*, pp. 70-78, March 1998.

- [8] C. Leangsuksun, J. Potter, and S. Scott, "Dynamic task mapping algorithms for a distributed heterogeneous computing environment," 4th *Heterogeneous Computing Workshop (HCW'95)*, pp.30-34, Apr. 1995.
- [9] 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.
- [10] T. Braun, H. J. Segel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, and B. Yao, "A taxonomy for describing matching and scheduling heuristics for mixed-machines heterogeneous computing systems," *Workshop on Advances in Parallel and Distributed Systems(APADA)*, West Lafayette, IN, pp.330-335, Oct. 1998.
- [11] A. H. Alhusaini, V. K. Prasanna, and C. S. Raghavendra, "A framework for mapping with resource co-allocation in heterogeneous computing system," 9th *Heterogeneous Computing Workshop (HCW 2000)*, pages 273-286, May 2000.
- [12] A. H. Alhusaini, V. K. Prasanna, and C. S. Raghavendra, "Run-Time Adaptation for Grid environments," *Proceedings 15th International Parallel and Distributed Processing Symposium*, pp. 864-874, April 2001.
- [13] H. Topcuoglu, S. Hariri, and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. on Parallel and Distributed System*, vol.13, pp.260-274, Mar. 2002.
- [14] A. H. Alhusaini, V. K. Prasanna, and C. S. Raghavendra, "A unified resources scheduling framework for heterogeneous computing environment," 8th *Heterogeneous Computing Workshop (HCW '99)*, pp.156-165, April 1999
- [15] N. Christofides, *Graph theory: An algorithmic approach*, Academic Press, 1975
- [16] 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,“ Intl. Workshop in Quality of Service, 1999.

