To the Graduate Council:

I am submitting herewith a dissertation written by Zhiao Shi entitled "Scheduling tasks with precedence constraints on heterogeneous distributed computing systems." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack J. Dongarra

Major Professor

We have read this dissertation
and recommend its acceptance:

Michael W. Berry

Donald W. Bouldin

James S. Plank

Accepted for the Council:

Linda Painter

Interim Dean of Graduate Studies

(Original signatures are on file with official student records.)

SCHEDULING TASKS WITH PRECEDENCE CONSTRAINTS ON HETEROGENEOUS DISTRIBUTED COMPUTING SYSTEMS

A Dissertation
Presented for the
Doctor of Philosophy Degree
The University of Tennessee, Knoxville

Zhiao Shi
December 2006

Dedication

To my parents, with love and gratitude.

Acknowledgments

There are many people who contributed to the development of this thesis with either their personal or technical support, for which I want to express my appreciation here.

First I would like to express my deepest gratitude to my thesis advisor, Dr. Jack Dongarra, whose expertise, understanding, and patience added considerably to my graduate experience. I thank him for his patience and encouragement that carried me through the course of this research and for his insights and suggestions that helped to shape my research skills.

I would also like to thank other members of my committee, Dr. Michael Berry, Dr. Don Bouldin and Dr. James Plank for providing valuable comments and constructive suggestions towards improving the quality of this research.

I would like to thank all my friends and research staff at Innovative Computing Laboratory (ICL) for providing a vibrant working environment. I especially thank Sudesh Agrawal, Zizhong Chen, Eric Meek, Kiran Sagi, Keith Seymour, Fengguang Song, Asim YarKhan, Haihang You, Sathish Vadhiyar, Scott Wells for their valuable help and precious friendship.

I would like to give particular thanks to Dr. Emmanuel Jeannot for his valuable advice and discussion about the details of the research. I am also grateful to Dr. Bing Zhang for his suggestion of using a bioinformatics application to test one of the proposed scheduling algorithms.

Last but not least, my sincerest thanks go to my parents for their unconditional love

and support during every stage of my life. Without them this thesis would never have come into existence.

The author acknowledges the support of the research by the National Science Foundation under Contract CCR-0331645 and CNS-0437508.

Abstract

Efficient scheduling is essential to exploit the tremendous potential of high performance computing systems. Scheduling tasks with precedence constraints is a well studied problem and a number of heuristics have been proposed.

In this thesis, we first consider the problem of scheduling task graphs in heterogeneous distributed computing systems (HDCS) where the processors have different capabilities. A novel, list scheduling-based algorithm to deal with this particular situation is proposed. The algorithm takes into account the resource scarcity when assigning the task node weights. It incorporates the average communication cost between the scheduling node and its node when computing the Earliest Finish Time (EFT). Comparison studies show that our algorithm performs better than related work overall.

We next address the problem of scheduling task graphs to both minimize the makespan and maximize the robustness in HDCS. These two objectives are conflicting and an ϵ-constraint method is employed to solve the bi-objective optimization problem. We give two definitions of robustness based on tardiness and miss rate. We also prove that slack is an effective metric to be used to adjust the robustness. The overall performance of a schedule must consider both the makespan and robustness. Experiments are carried out to validate the performance of the proposed algorithm.

The uncertainty nature of the task execution times and data transfer rates is usually neglected by traditional scheduling heuristics. We model those performance characteristics of the system as random variables. A stochastic scheduling problem is formulated to

minimize the expected makespan and maximize the robustness. We propose a genetic algorithm based approach to tackle this problem. Experiment results show that our heuristic generates schedules with smaller makespan and higher robustness compared with other deterministic approaches.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Heterogeneous Distributed Computing Systems (HDCS), including the recently advocated Grid computing platform, utilize a distributed set of high performance machines, connected with high speed networks to solve computationally intensive applications coordinately [38, 41, 42]. Applications usually consists of a set of tasks, with or without dependencies among them.

One of the most important components for achieving high performance with HDCS is the *mapping* strategies they adopt. Mapping of an application involves the matching of tasks to machines and scheduling the order of execution for these tasks [20]. We will use the terms *mapping* and *scheduling* interchangeably without too much confusion. In general, the scheduling problem is computationally intractable even under simplified assumptions [44]. Many heuristics thus have been proposed [64, 65]. The complexity of the problem increases when the application is executed in a HDCS due to the fact

that the processors and network connections in the system may not be identical and it takes different amounts of time to execute the same task or transfer the same amount of data.

1.1 Motivation

In this thesis, an application is modeled by a task graph. A task graph is a directed acyclic graph (DAG) in which nodes represent tasks and edges represent the data dependencies among the tasks. Although there are many heuristics proposed for scheduling DAG-type applications, most of them assume that the processors are equally capable, i.e. each processor can execute all the tasks. In real world, this assumption usually does not hold, especially in the case of HDCS. An HDCS such as the Grid system can be composed of processors with wide varieties of types, processing power and capabilities. For example, a GridSolve system [9] typically consists of an agent and multiple servers. Any service provider can connect his server to an existing agent. Since typically the servers in a GridSolve system are set up by different service providers, they usually have distinctive processing capabilities (software). Scheduling with traditional heuristics in this type of system can be inefficient since they do not consider the effect of the processors' different capabilities. Clearly there is a need for a scheduler that acknowledges this important fact.

Traditional DAG scheduling algorithms assume that the system is time-invariant, where the exact performance of the system can be known *a priori*. For example, it is

assumed that the execution time of a task can be estimated and does not change during the course of execution. However, due to the resource sharing among multiple users in HDCS, the performance of the system can vary during the execution of the application. Under this condition, there is a need to find schedules that are less vulnerable to the variance of system performance, i.e. more robust. Minimizing the schedule length (makespan) based on the estimated system performance is not enough, since short makespan does not necessary guarantee a small turn-around time in a real computing environment. A good schedule must also be robust. As we will see, minimizing the makespan and maximizing the robustness of a schedule is two conflicting factors. There can be a trade-off between these two objectives.

Due to the non-determinism of system performance, scheduling with stochastic information of the performance characteristics can be useful. For instance, task execution times or data transfer rates can be modeled with random variables. Each random variable takes on some values according to its probabilistic distribution. Previously, only mean values are used for scheduling. Such approaches do not consider the temporal heterogeneity of the resources. It has been shown that they can lead to inferior schedules. We need to come up with an algorithm that utilizes the stochastic information about the task execution times and data transfer rate to produce a better schedule in terms of minimizing the makespan and maximizing the robustness.

1.2 Contributions

In this dissertation, we developed several scheduling algorithms to address the problem of producing efficient schedules in HDCS. The contributions of this research are summarized as follows:

1. A framework for evaluating different scheduling algorithms has been developed for comparing our algorithms with other existing algorithms. A random task graph generator is designed to generate task graphs with specific parameters for the performance study. It can generate many types of graphs in order to perform unbiased comparisons of different algorithms.

2. We proposed a list scheduling-based algorithm that takes into account the different capabilities of the available processors. When assigning task node weights, the algorithm considers the effect of a task's scarcity of capable processors. A task with a small percentage of capable processors is given higher weight because it is more urgent to schedule such task. A new method for calculating the Earliest Finish Time (EFT) is proposed to incorporate the average communication cost between the current scheduling task and its children.

3. The robustness of a schedule measures the degree to which the schedule is insensitive to the disturbances in task execution times. It should reflect how stable the actual makespan will be with respect to the expected value. We proposed two definitions of robustness for a schedule based on the relative tardiness and miss

rate.

4. We designed a genetic algorithm based scheduling heuristic to address the problem of robust task scheduling. Slack is identified as an important metric that is closely related to the robustness of a schedule. In order to optimize both the robustness and makespan, we include slack and makespan as the fitness value of each individual in the population. We show that maximizing the robustness or slack and minimizing the makespan are two conflicting objectives. A bi-objective optimization problem is formulated. The proposed algorithm is flexible in finding the best solution in terms of the overall performance considering both makespan and robustness.

5. We further developed a genetic algorithm based heuristic for the stochastic scheduling problem. This involves the calculation of makespan distribution where all the task execution times and data transfer rates are modeled as random variables with certain probability distributions. A procedure for estimating the makespan distribution of a task graph with great accuracy is presented. Our heuristic generates schedules with a smaller makespan and higher robustness compared with other deterministic approaches.

1.3 Outline of the dissertation

The rest of the dissertation is organized as follows. Chapter 2 provides relevant background information about task scheduling in HDCS. An overview of the DAG scheduling

problem is presented first. Then, different existing scheduling algorithms are categorized and briefly reviewed. In Chapter 3, we propose an algorithm for scheduling task graphs in computing systems where processors have different capabilities. We describe the robust task scheduling algorithm in Chapter 4. In Chapter 5 we address the problem of scheduling task graphs with stochastic information about the system performance. Chapter 6 discusses future work and concludes the dissertation.

Chapter 2

Review of Literature

2.1 Introduction

A *distributed system* is a computing platform where hardware or software components located at networked computers communicate and coordinate their actions only by passing messages [26]. It enables users to access services and execute applications over a heterogeneous collection of computers and networks. Heterogeneity applies to networks, computer hardware, operating systems etc. In this research, we will refer to such a system as *heterogeneous distributed computing system* (HDCS). The term, *distributed computing*, usually refers to any system where many resources are used to solve a problem collaboratively. In recent years, HDCS has emerged as a popular platform to execute computationally intensive applications with diverse computing needs.

The problem of *mapping* (including *matching* and *scheduling*) tasks and communications is a very important issue since an appropriate mapping can truly exploit the

parallelism of the system thus achieving large speedup and high efficiency [19]. It deals with assigning (matching) each task to a machine and ordering (scheduling) the execution of the tasks on each machine in order to minimize some cost function. The most common cost function is the total schedule length, or *makespan*. In this review, we will use mapping and scheduling interchangeably. Unfortunately, the scheduling problem is extremely difficult to solve and is proved to be *NP-complete* in general. Even problems constructed from the original mapping problem by making simplified assumptions still fall in the class of NP-hard problems. Consequently, many heuristics have been proposed to produce adequate yet sub-optimal solutions. In general, the objective of task scheduling is to minimize the completion time of a parallel application by properly mapping the tasks to the processors. There are typically two categories of scheduling models: *static* and *dynamic* scheduling. In the static scheduling case, all the information regarding the application and computing resources such as execution time, communication cost, data dependency, and synchronization requirement is assumed to be available *a priori* [53, 65] . Scheduling is performed before the actual execution of the application. Static scheduling offers a global view of the application thus usually generates high quality schedules. On the other hand, in the dynamic mapping a more realistic assumption is used. Very little *a priori* knowledge is available about the application and computing resources. Scheduling is done at run-time. In order to support load balancing and fault tolerance, tasks can be reallocated during the execution. In this research, we focus on static scheduling.

2.1.1 The application model

Certain computational problems can be decomposed into a large number of tasks that can be executed in any order, such as parameter sweep applications. These tasks are mutually independent, i.e. there is no precedence constraint among them. Given a set of independent tasks and a set of available resources, independent task scheduling attempts to minimize the total execution time of the task set by finding an optimal mapping of tasks to machines.

Another popular parallel application model is the *task precedence graph* model [65]. In this model, an application can be represented by a *Directed Acyclic Graph* (DAG). In a DAG, nodes represent the tasks and the directed edges represent the execution dependencies as well as the amount of communication between the nodes.

Specifically, a parallel application is modeled by a DAG $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ represents the set of n tasks to be executed, and the set of e weighted, directed edges E represents communication requirement between tasks. A node in the DAG represents an atomic task that is a set of instructions that must be executed sequentially without preemption on the same processor. The weight of the node reflects the amount of work associated with the task. Usually, in a HDCS, the execution time of a task is different for each processor in the system. A typical way of setting the weight of the task node is to use the average execution time among all the processors in the HDCS. The edges in the DAG correspond to the data transfer and precedence constraints among the nodes. The weight of an edge is the data size between the two

tasks connected by the edge. Thus, $e_{i,j} = (v_i, v_j) \in E$ indicates communication from task v_i to v_j, and $|e_{i,j}|$ represents the volume of data sent from v_i to v_j. The source node of an edge is called the *parent* node while the sink node is called the *child* node. A node without a parent is called an *entry* node, and a node without a child is called an *exit* node. Fig. 2.1 shows an example of the task precedence graph. In the example, v_2 is the parent of v_4 and v_5, v_6 is the child of v_3, v_4 and v_5, v_1 is an entry node and v_6 is an exit node. The number next to each edge denotes the data volume to be transferred from the source node to the sink node. In this research, we focus on the applications which can be modeled as DAGs. Furthermore, we assume the following: (1) Tasks are non-*preemptive*: in a preemptive resource environment, a running task can be preempted from execution. Preemption is commonly used in priority-based or real-time systems. For example, when a pending task A's deadline is approaching, it is necessary to preempt one running job B, whose deadline is not as imminent as A's and assign the resource to task A. As a result of preemption, the scheduling is very complicated. In a non-preemptive computational resource, preemption is not allowed, i.e., once a task is started on such a resource, it cannot be stopped until its completion. (2) Only process level parallelism is considered. The application consists of a set of tasks (processes). Each task can only be assigned to one processor.

2.1.2 The HDCS model

As mentioned previously, the target resource environment for the DAG scheduling is a HDCS. It consists of a network of m *processing elements*. Each processing element

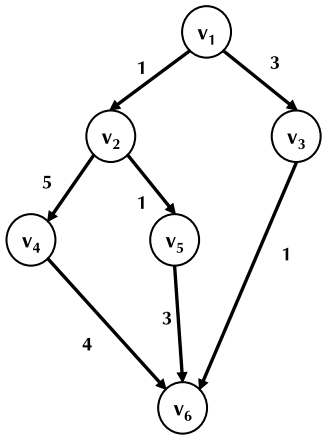


Figure 2.1: A task precedence graph

includes a CPU and local memory. Communication between processors is through a message passing mechanism. The processors are heterogeneous which means they have different processing speed. In addition, processors are *unrelated*, i.e. if a machine p_i has a lower execution time than a machine p_j for a task v, then the same is not necessary true for any task v'. This happens when the HDCS is composed of many machines with diverse architectures and there is a variety of different computational needs among the tasks. Therefore, the way in which a task's needs correspond to a machine's power may differ for each possible combination of tasks and machines. Furthermore, it is possible that not every task can be executed on each machine in the HDCS. This indicates that each machine can have different capabilities. The processing elements are connected by a network of certain topology. One of the important characteristic of the HDCS is whether the performance of the environment is deterministic. For example, if it is expected that computing task v_i on processor p_j takes t time units, what is the real

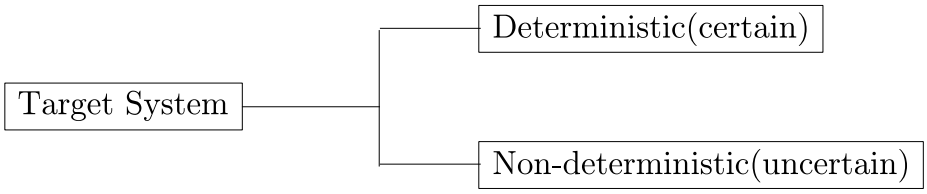


Figure 2.2: Target system taxonomy of scheduling algorithms

value for t? Usually it is important for scheduling algorithms to take into account this uncertainty, especially in a highly shared environment. Thus, on the highest level, we divide the scheduling algorithms into two categories according to the certainty of target computing system's performance prediction. (See Fig. 2.2)

2.2 Static DAG scheduling in deterministic HDCS

Kwok and Ahmad give a survey of various static DAG scheduling algorithms in [65]. The authors classify the considered algorithms into different categories based on the assumptions used in the algorithms such as the task graph structure (arbitrary DAG or restricted structure such as trees), computation costs (arbitrary costs or unit costs), communication (communication cost considered or not), duplication (task duplication allowed or not), number of processors (limited or unlimited) and connection type among the processors (fully connected or arbitrary connected). However, the 27 algorithms surveyed are mainly designed for a homogeneous environment. For algorithms designed for heterogeneous systems, there are basically four types, namely list scheduling based algorithms, clustering heuristics, task duplication heuristics and random search based

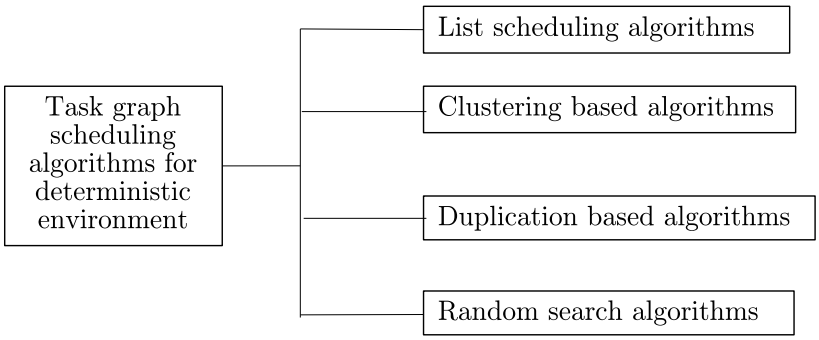


Figure 2.3: Taxonomy of static task scheduling algorithms for deterministic environment

algorithms [98] (see Fig. 2.3).

2.2.1 List scheduling

List scheduling is a class of scheduling algorithms that assign tasks one by one according to their priorities [65]. The essence of list scheduling is to make an ordered task list by assigning tasks some priorities and then repeatedly perform the following two steps until all the tasks in the list are scheduled:

1. Remove the first task from the list;

2. Allocate the task to a processor that will optimize some predefined cost function

The pseudo-code of list scheduling is presented in Alg. 2.2.1.

Algorithm 2.2.1: LIST SCHEDULING()

Calculate the priority of each task according to some predefined formula
PriorityList $= \{v_1, v_2, ..., v_n\}$ is sorted by descending order of task priorities
while *PriorityList* is not empty
  Remove the first task from the *PriorityList* and assign
   it to an appropriate processor in order to optimize a predefined cost function
return (*schedule*)

There are two important questions in a list scheduling algorithm: (1) How to compute a task node's priority? (2) How to define the cost function? The first question is related to the way the algorithm views the node's urgency of being scheduled. In the earlier list scheduling algorithms, the target computing systems are generally homogeneous. Some algorithms do not take into account the communication costs. Level-based heuristics are proposed for this case. For example, in the HLEFT algorithm [1], the level of a node denotes the sum of computation costs of all the nodes along the longest path from the node to an exit node. For task scheduling in HDCS the communication cost usually cannot be ignored. In addition, the execution time of the same task will differ on different resources as well as the communication cost via different network links. The priority of a node depends on where it will be allocated. However, the priority must be set before any allocation decision can be made. In order to avoid this dilemma, approximations of task node weight (computation cost) and edge weight (communication cost) are used. The approximation can be based on the average value among the processors (resp. links), the best value, or the worst value, etc. There is no decisive conclusion on which one should be used [109]. Two important attributes used to calculate the priority of a task node are the *t-level* (top level) and *b-level* (bottom

level) [64, 65]. The *t-level* of a node is defined as the length of a longest path from an entry node to the node (excluding the node itself). The length of a path is the sum of all the node and edge weights along the path. As pointed out previously, the weights are approximations based on one of the criteria. The *t-level* is related to the *earliest start time* of the node. The *b-level* of a node is the length of a longest path from the node to an exit node. The *critical path* of a DAG is a longest path in the DAG. Clearly, the upper bound of a node's *b-level* is the critical path of the DAG. *B-level* and *t-level* can be computed with time complexity $O(e + n)$, where e is the number of edges and n is the number of nodes in the DAG. The second question deals with the selection of "best" processor for a task. In homogeneous systems, a commonly used cost function is called *earliest start time* [51]. For example, the Earliest Time First (ETF) algorithm computes, at each step, the earliest start times for all ready nodes and then selects the one with the smallest earliest start time. When two nodes have the same value of their earliest start times, the ETF algorithm breaks the tie by scheduling the one with the higher static level. While scheduling in HDCS, many other cost functions are proposed. In the following, we describe a few related list scheduling algorithms.

Modified Critical Path (MCP)

The MCP algorithm [106] assigns the priority of a task node according to its *As Late As Possible* (ALAP) value. The ALAP time of a node is defined as $ALAP(n_i) = T_{critical} - b\_level(n_i)$, where $T_{critical}$ is the length of the critical path. The ALAP of a task node is a measure of how far the node's start time can be delayed without increasing

the schedule length. The MCP algorithm first computes the ALAPs of all the nodes, then sorts the nodes by ascending order of ALAP times into a list. Ties are broken by using the smallest ALAP time of the successor nodes, the successors of the successor nodes, and so on. Then it schedules the first node in the list to the processor that allows the earliest start time, considering idle time slots. After the node is scheduled, it is removed from the list. This step is repeated until the list becomes empty. The time complexity of MCP is $O(n^2 \log n)$.

Dynamical Level Scheduling (DLS)

The Dynamic Level Scheduling algorithm [94] is one of the earliest list scheduling algorithms designed for heterogeneous computing systems. However, in contrast to traditional list scheduling, DLS does not maintain a scheduling list during the scheduling process. It determines node priorities dynamically by assigning an attribute called the *dynamic level* (DL) to all unscheduled nodes at each scheduling step. The DL of task node n_i on processor p_j, denoted as $DL(n_i, p_j)$, reflects how well n_i and p_j are matched. The DL is determined by two terms. The first term is the static level (SL) of the node, which is defined as the maximum sum of computation costs along a path from n_i to an exit node. This is also called *static b-level*. Notice that the static b-level does not include the communication cost along the path. The second term is *start time* (ST) of n_i on p_j. The DL is defined as $SL(n_i) - ST(n_i, p_j)$. At each scheduling step, the DLS algorithm computes the DL for each ready node on every processor. Then, the node-processor pair that constitutes the largest DL among all other pairs is selected so

that the node is scheduled to the processor. This process is repeated until all the nodes are scheduled.

The algorithm is adapted for scheduling in a heterogeneous environment by modifying the definition of DL. A term $\Delta(n_i, p_j) = E^*(n_i) - E(n_i, p_j)$ is added to the expression of DL to account for the varying processing speed, where $E^*(n_i)$ is the median of execution times of n_i over all processors. In order to consider how the descendants of n_i matches p_j, another term called *descendant consideration* (DC) is added to the DL expression. The DC term is the difference between the median execution time of the most significant descendant $D(n_i)$ to which n_i passes the most data and a lower bound on the time required to finish execution of $D(n_i)$ after n_i finishes execution on p_j. This reveals how well the most "expensive" descendant of n_i match p_j if n_i is scheduled on p_j. In addition to the descendant consideration effect, the algorithm takes into account the situation where certain processors are not capable of executing some task nodes. The algorithm defines the cost of not scheduling node n_i on its preferred processor where the DL is maximized as the difference between the highest DL and the second highest DL for n_i over all processors. This term is added to DL. The time complexity of DLS algorithm is $O(n^3 m f(m))$, where $f(m)$ is the function used to route a path between two given processors on the targeted system.

Dynamical Critical Path (DCP)

In [63], Kwok et al. proposed a list scheduling heuristic that includes the following features: (1) Instead of maintaining a static scheduling list, the algorithm selects the

next task node to be scheduled dynamically. It assigns dynamic priorities to the nodes at each step based on the *dynamic critical path* (DCP) so that the schedule length can be reduced monotonically. As defined previously, the critical path of a task graph is a set of nodes and edges, forming a path from an entry node to an exit node, of which the sum of computation costs and communication costs is maximized. The critical path changes during the scheduling process. The critical path at an intermediate scheduling step is called dynamic critical path. In order to reduce the length of the DCP at each scheduling step, the node selected for scheduling is the one that has no unscheduled parent node on the DCP. (2) During the processor selection step, the algorithm employs a start time look-ahead strategy. Suppose n_i is considered for scheduling. Let n_c be the child node of n_i, which gives the smallest difference between its upper bound and lower bound start time. Then n_i should be scheduled to the processor that minimizes the sum of lower bounds of n_i and n_c's start time. It considers both the current node and its critical child node when scheduling aiming to avoid scheduling the node onto an inappropriate processor. In addition, the algorithm does not exhaustively examine all processors for a node. Instead, it only considers the processors that hold the nodes that communicate with this node. The algorithm has a time complexity of $O(n^3)$.

Heterogeneous Earliest Finish Time (HEFT)

In [98], Topcuoglu et al. presented a heuristic called Heterogeneous Earliest Finish Time (HEFT) algorithm. The HEFT algorithm sets the weight of a task node as the average execution cost of the node among all available processors. Similarly, the weight

of an edge is the average communication cost among all links connecting all possible processor pair combinations. The priority of a task node is the *b-level* of that node, which is based on mean computation and mean communication costs. The task list is created by sorting the tasks in decreasing order of *b-level*. Ties are broken randomly. The selected task is then assigned to the processor which minimizes its earliest finish time with an insertion-based approach that considers the possible insertion of a task in an earliest idle time slot between two already-scheduled tasks on the same resource. The time complexity of HEFT is $O(e \times m)$, where e is the number of edges and m is the number of processors. The HEFT algorithm is considered one of the best algorithms for scheduling tasks onto heterogeneous processors.

Some other list scheduling based algorithms include Mapping Heuristic (MH) [35], Fast Critical Path (FCP) [82], and Insertion Scheduling Heuristic (ISH) [61].

2.2.2 Clustering based heuristics

Another class of DAG scheduling algorithms is based on a technique called *clustering* [46, 59, 107]. The basic idea of clustering based algorithm is to group heavily communicated tasks into the same cluster. Tasks grouped into the same cluster are assigned to the same processor in an effort to avoid communication costs. There are basically two types of clusters; *linear* and *nonlinear*. Two tasks are called independent if there are no dependence paths between them. A cluster is called nonlinear if there are two independent tasks in the same cluster, otherwise it is called linear. Fig. 2.4 shows examples of linear and nonlinear clustering. In Fig. 2.4(a), three clusters are created.

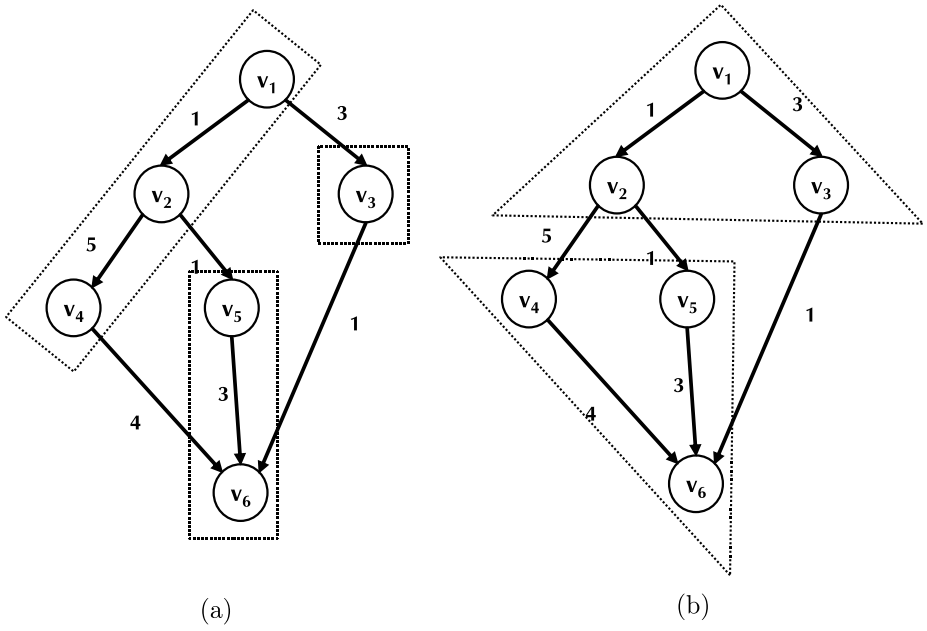


Figure 2.4: (a) linear clustering (b) nonlinear clustering of Fig. 2.1

The task node(s) in each cluster is dependent on each other. In contrast, in cluster $\{v_1, v_2, v_3\}$ of Fig. 2.4(b), task v_2 and v_3 are independent. Similarly, task v_4 and v_5 are independent in the other cluster.

There are essentially two steps in a clustering based heuristic; grouping the nodes into clusters and mapping the clusters to processors. Initially, each task is assumed to be in a separate cluster. Then a series of refinements are performed by merging some existing clusters. A final clustering will be derived after certain steps. In order to avoid high time complexity, once the clusters have been merged they cannot be unmerged in the subsequent steps. During the mapping phase, a sequence of optimizations are carried out: (1) cluster merging: It is possible that the number of clusters is greater than

the number of processors. Then it is necessary to further merge the clusters; (2) task ordering: if the tasks in a cluster are related by precedence constraints, the execution order of the tasks is arranged based on such constraints. Next, we briefly introduce some clustering heuristics.

Linear Clustering Method (LC)

In [59], Kim and Browne proposed a linear clustering algorithm. The clustering phase of the algorithm is done in the following fashion. Initially all edges are marked unexamined. The following steps are performed: (1) Determine the longest path CP composed of only unexamined edges, by using a cost function. The authors used a weighted combination of communication costs and computation costs along the path as the cost function. For simplicity, the cost function can be the length of a critical path that is the sum of all communication costs and computation costs along the path as defined previously. Nodes in this path are grouped into a cluster and their edge costs are zeroed. (2) Mark all edges incident to the nodes in CP as examined. (3) Recursively apply steps 1 and 2 until all edges are examined. In this way, a set of linear clusters are derived. After the clustering phase, the algorithm attempts to merge two or more linear clusters into one in order to balance the workload of processors and reduce both the number of resources to be used and interprocessor communication cost.

Once the clustering phase is complete, a task graph is transformed into a *virtual architecture graph* (VAG). The VAG represents an optimal multiprocessor architecture for the task graph. A *Dominant Request Tree* (DRT), which is a maximal spanning

tree of a VAG, is then constructed. In the case of mapping onto heterogeneous systems, it is important to utilize resources with high computing power as much as possible. In order to get the information about resources, a prescanning of architecture graphs prior to physical mapping is necessary. This is done by creating a *Dominant Service Tree* (DST) with a maximal spanning tree algorithm. The basic idea of mapping the clusters onto the processors is to find a subgraph isomorphism from DRT to DST that can minimize the total makespan. The problem of tree-to-tree mapping is solved by exploiting sequential mapping the order of nodes determined during construction of a DRT and node information to avoid exhaustive matching between two trees.

The complexity of LC algorithm is $O(n(n+e))$. For a dense graph where $e = O(n^2)$, the complexity becomes $O(n^3)$.

Edge Zeroing algorithm (EZ)

The Edge Zeroing algorithm (EZ) [88] uses edge weights as the criteria to merge clusters. Two clusters can be merged only if the merging does not increase the makespan. The algorithm can be described as the follows:

(1) Sort the edges of the task graph in order of descending weight.

(2) Zero the highest edge weight if the makespan does not increase. Two clusters are merged so that all edges incident on these two clusters are also zeroed out. The ordering of nodes in the cluster is based on their *static b-level*s.

(3) Repeat step (2) until all edges are scanned.

The complexity of EZ algorithm is $O(e(n+e))$.

Mobility Directed algorithm (MD)

The MD algorithm [106] defines a new attribute called *relative mobility* for a task node. It is computed as follows:

$$\frac{CP\_length - (b\text{-}level(n_i) + t\text{-}level(n_i))}{w(n_i)} \tag{2.1}$$

where $w(n_i)$ is the weight of task node n_i.

The algorithm works as follows: First, the relative mobility of each node is calculated. Let L be the set of all unexamined nodes. Initially, all task nodes are in L. Next, Let L' be the group of nodes in L that have minimal relative mobility. Let $n_i \in L'$ be a node without any predecessors. If n_i can be scheduled onto the first processor, then schedule it onto the first processor. Otherwise keep trying the next processor until a processor that can accommodate n_i is found. The criteria for a processor being able to accommodate n_i is whether the *moving interval* of n_i, defined as the time interval from the *as-soon-as-possible (ASAP)* start time and *as-late-as-possible (ALAP)* start time [65] interferes with those of the nodes already scheduled on that processor. If it does not interfere, then the processor can accommodate n_i. Suppose n_i is scheduled on processor j, then the weights of edges connecting n_i and all other nodes already scheduled on processor j are changed to zero. If n_i is scheduled before n_k on processor j, add an edge with weight zero from n_i to n_k. Similarly, if n_i is scheduled after n_j

on the processor, then add an edge with weight zero from n_j to n_i. This step ensures that no loop is formed. If so, schedule n_i to the next available space. Then, recalculate the relative mobilities of the new graph. Remove n_i from L and repeat the above steps until L becomes empty. The complexity of this algorithm is $O(n^3)$.

Dominant Sequence Clustering (DSC)

Yang and Gerasoulis [107] proposed a clustering algorithm called *Dominant Sequence Clustering* (DSC). The *dominant sequence* (DS) of a clustered DAG is the longest path of the scheduled DAG. In contrast, the *critical path* (CP) is the longest path of a clustered but not scheduled DAG. The main idea behind a dominant sequence heuristic is to identify the *DS* at each step and then zero edges in that *DS*. We briefly describe the algorithm as follows:

(1) Initially, all task nodes are unexamined and put into set *UNS*. A free node list L containing all nodes whose predecessors have been scheduled is constructed. Compute *b-level* for all the nodes and *t-level* for each free node.

(2) Let the first node of L be n_i. Two cases are considered.

- n_i is a node on the DS. If zeroing the edge between n_i and one of its parents leads to a minimal *t-level* of n_i, then zero that edge. If no such zeroing is accepted, the node will remain in a single node cluster.
- n_i is not a node on the DS. If zeroing the edge between n_i and one of its parents can minimize the *t-level* of n_i under the following constraint, then zero

that edge. The constraint mandates that zeroing incoming edges of a free node should not affect the future reduction of *t-level* of n_j, where n_j is a not-yet free node with a higher priority, if the *t-level* of n_j is reducible by zeroing an incoming DS edge of n_j. If some of n_i's parents are entry nodes with no child other than n_i, merge part of those parents so that the *t-level* of n_i is minimized. If no zeroing is accepted, n_i remains in a single node cluster.

(3) Recompute the *t-level* and *b-level* of the successors of n_i and remove n_i from *UNS*.

(4) Repeat steps (2)-(3) until *UNS* becomes empty.

The complexity of DSC algorithm is $O((e+n)\log n)$.

2.2.3 Task duplication based heuristics

The basic idea behind task duplication based (TDB) scheduling algorithms is to use the idle time slots on certain processors to execute duplicated predecessor tasks that are also being run on some other processors, such that communication delay and network overhead can be minimized [2, 3, 28, 61, 62, 78, 83, 84]. In this way, some of the more critical tasks of a parallel program are duplicated on more than one processor. This can potentially reduce the start times of waiting tasks and eventually improve the overall completion time of the entire program. Duplication based scheduling can be useful for systems having high communication latencies and low bandwidths.

There are basically two types of TDB existing in the literature: *Scheduling with Partial Duplication (SPD)* and *Scheduling with Full Duplication (SFD)* [78]. In an SPD

algorithm, the parent of a *join node* is not duplicated unless it is critical. A join node is defined as a node with in-degree greater than one (i.e., a node with more than one immediate predecessors). Algorithms in this category need to find the *critical immediate predecessor (CIP)* of the node n_i to be scheduled. The CIP of a node n_i is defined as the immediate predecessor (n_j) that provides the largest message arriving time from n_j to n_i. The message arriving time is the time the message from n_j arrives at n_i. Then the join node is scheduled on the processor where the CIP has been scheduled. Due to the limited task duplication, algorithms in this category have a low complexity but may not be useful for systems with high communication overhead. On the other hand, SFD algorithms attempt to duplicate all the parents of a join node and apply the task duplication algorithm to all the processors that have any of the parents of the join node. Thus, algorithms in this category have a higher complexity but typically show better performance than SPD algorithms. In the following, we will describe some TDB scheduling algorithms.

Duplication Scheduling Heuristic (DSH)

The Duplication Scheduling Heuristic (DSH) algorithm [61] uses the idea of list scheduling combined with duplication to reduce the makespan. Task nodes are given priorities to indicate the urgency of being scheduled. The algorithm uses the *static b-level* as the node's priority. In selecting a processor for a node, the algorithm first calculates the start time of the node on the processor without duplication of any predecessor. Next, it duplicates the predecessors of the node into the idle time slot of the processor until

either such slot is unavailable or the start time of the node does not improve. The algorithm works as follows:

(1) Compute the *static b-level* for each node and set it as its priority. Sort the nodes into a list L in the order of decreasing priority.

(2) Pick the first task node n in L.

(3) For each processor P, perform the following steps:

  (i) Compute the start time of n on P, ST. Set the candidate as n.

  (ii) Consider the set of immediate predecessors of the candidate n. Let n' be the immediate predecessor that is not scheduled on P and whose message for a candidate has the latest arrival time. Duplicate n' into the earliest idle time that can accommodate it on P.

  (iii) If such a time slot is unavailable, then ST is recorded and go to step (3). Otherwise, the candidate's start time is replaced by the new start time if the new start time is smaller. Set the candidate as n'. Go to step (3ii).

(4) Schedule n to the processor that gives the smallest ST and perform all necessary duplications on that processor.

(5) Repeat steps (2)-(4) until all the nodes are scheduled.

The time complexity of the DSH algorithm is $O(n^4)$.

Critical Path Fast Duplication (CPFD)

Ahamd and Kwok [2, 3] proposed a duplication based algorithm called *Critical Path Fast Duplication* (CPFD). The authors believe that selecting the "important" nodes for duplication is key to obtaining a short makespan. They classified the task nodes in a DAG into 3 categories in the order of decreasing importance: *Critical Path Node*s (*CPN*), *In-Branch Node*s (*IBN*) and *Out-Branch Node*s (*OBN*). CPNs are on a critical path. These are most important because their finish times effectively determine the final makespan. An IBN node is a node that is not a CPN and from which there is a path reaching a CPN. The IBNs are also important because timely scheduling of these nodes can help reduce the start times of the CPNs. An OBN is a node that is neither a CPN nor an IBN. The OBNs are relatively less important because they usually do not affect the makespan.

The algorithm has the following steps:

(1) Determine a Critical Path. Ties are broken by selecting the one with a larger sum of computation costs. Based on the importance of a node, a priority list called *CP-Dominant Sequence* is constructed in a way that CPNs can be scheduled as soon as possible. In addition, precedence constraints are also preserved.

(2) Select the first unscheduled CPN in the CPN-Dominant Sequence as the candidate n_c.

(3) Let P_SET be a set of processors, including all the processors holding the candi-

date's parent nodes and an empty processor.

(4) For each processor P in P_SET, find the *Earliest Start Time (EST)* of the candidate on P and record it.

(5) Schedule the candidate to the processor P that gives the smallest value of EST. All necessary duplications are performed.

(6) Repeat the process from step (2) to step (5) for each OBN with P_SET containing all the processors in use together with an empty processor. The OBNs are considered one by one in topological order.

(7) Repeat step (2)-(6) until all CPNs are scheduled.

The process of determining the candidate n_c's EST on processor P works as follows: Let the start time of n_c on P be ST. Consider the set of candidate's immediate predecessors. Let n be the immediate predecessor that is not scheduled on P and whose message for n_c has the latest arrival time. Try to duplicate n on the earliest idle time slot that can accommodate it on P. If the duplication is successful and the new start time of n_c is smaller than ST, then let ST be the new start time. Now set the candidate to n and repeat from the beginning until the duplication is unsuccessful. At this point, the value of ST is the EST of n_c on P. The time complexity of the CPFD algorithm is $O(n^4)$.

Duplication First and Reduction Next (DFRN)

The two algorithms introduced previously are both SFD algorithms. In [78], the authors noted that due to its high time complexity, SFD algorithms may not be suitable for task graphs with large number of nodes. They proposed an algorithm called *Duplication First* and *Reduction Next* (DFRN). An SFD algorithm recursively estimates the effect of a possible duplication and decides whether to duplicate each node one by one. Consequently, for a DAG with n nodes, each node may be considered n times for duplication in the worst case. In contrast, the DFRN algorithm first duplicates all parent nodes in a bottom-up fashion to the parent that has been scheduled on the same processor, without estimating the effect of their duplications. Then each duplicated task is removed if the task does not meet certain conditions. DFRN applies the duplication only for the critical processor with the hope that the critical processor is the best candidate for the join node (node with multiple immediate predecessor). The *critical immediate parent* (CIP) of a join node n is the immediate parent whose message for the join node has the latest arrival time. The processor on which the CIP of n is scheduled is called the *critical processor* (CP) of n.

The algorithm is briefly described as follows:

(1) Set the priority of each node and sort them into a list L by descending priority. Any list scheduling algorithm is suitable for setting the priority.

(2) Consider the first node n_i in the list L. If n_i is not a join node, identify its *immediate parent* (IP) (n_{IP}). If n_{IP}, which is assigned on p_{IP}, is the *last node* (LN) (the most

recent assigned node on n_{IP}), schedule n_i on n_{IP}, otherwise copy the schedule up to n_{IP} onto an unused processor p_u and schedule n_i onto p_u. If n_i is a join node, identify its CIP n_{CIP} and critical processor p_c. If n_{CIP} is LN on p_c, then apply $DFRN(n_i, p_c)$, otherwise copy the schedule up to n_{CIP} onto p_u and apply $DFRN(n_i, p_u)$.

(3) The procedure $DFRN(n, p)$ first tries duplication with *try_duplication*(n, p) and then seeks to delete unnecessary duplication performed with *try_deletion*(n, p). In *try_duplication*(n, p), it duplicates the immediate parent that gives the largest *message arriving time* (MAT) to n. Then the procedure recursively searches its immediate parent from n in a bottom-up fashion until it finds the parent that has already been scheduled on p. When it finds the parent on p, it stops the search and duplicates the parents traversed so far on p. Next, in *try_deletion*(n, p), the procedure decides whether to delete any of the duplicated tasks based on the two conditions. The first condition considers the case where the output of a duplicated task is available earlier by a message from the task on another processor than the duplicated task itself. In this case, the duplication is unnecessary thus it is deleted. The second condition deals with a situation where the duplication does not decrease the earliest start time $EST(n, p)$.

(4) Remove n_i from L.

(5) Repeat steps (2)-(4) until L becomes empty.

The time complexity of the DFRN algorithm is $O(n^3)$.

2.2.4 Guided random search algorithms

The task scheduling problem is a search problem where the search space consists of an exponential number of possible schedules with respect to the problem size. *Guided random search* algorithms are a class of search algorithms based on enumerative techniques with additional information used to guide the search. They have been used extensively to solve very complex problems. A common characteristic of these algorithms is that they are stochastic processes with the use of random probability. *Evolution computation* and *stochastic relaxation* are the two major categories of guided random search algorithms. *Simulated annealing* [99] is one of the most important stochastic relaxation algorithms [45]. During the search process, it makes decisions about accepting or rejecting a random generated move based on a random probability related to an annealing temperature. It is able to explore the whole solution space that is independent from the initial starting point. Evolution computation is based on the natural selection principles. A *Genetic algorithm* (GA) [31, 47, 49] is one type of evolution computations that is commonly used. Its search sampling consists of a pool of potential solutions called *population* that is substantially different from other random search algorithms. It works with an encoding of the solutions, not directly with the solutions. In addition, it uses probabilistic transition rules to evolve from one generation of population to another.

Fig. 2.5 shows a cycle of a genetic algorithm. There are basically four stages. In each cycle, a new generation of candidate solutions for the problem considered is produced.

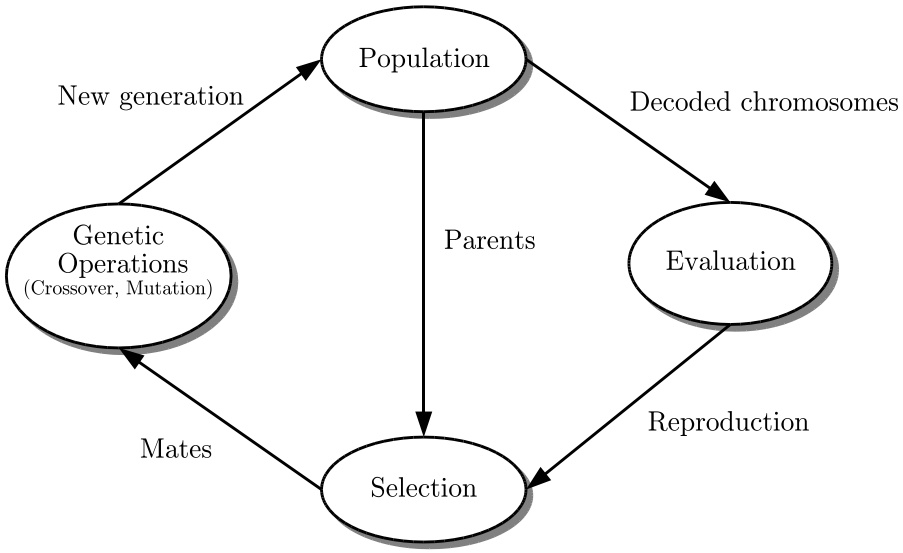


Figure 2.5: The cycle of genetic algorithms

At the first stage, an initial population is generated as the starting point of the search process. Each element of the population (chromosome) is encoded into a string. In the *evaluation* stage, each individual of the population is evaluated using an objective function to measure its fitness. Based on the fitness of each chromosome, a *selection* mechanism chooses mates for genetic operations. The selection policy ensures that the fittest chromosome has a greater probability to be chosen for mating. The applied genetic operators include *crossover* and *mutation*. These operators modify the structure of the involved chromosome to produce a new generation of population.

The basic operating structure of GA is the string. A typical string is composed of a sequence of characters of finite length λ over a problem specific vocabulary V. For

example, a string can be represented by:

$$S = a_1 a_2 ... a_\lambda \quad where \quad a_1, a_2, ..., a_\lambda \in V \tag{2.2}$$

Strings of current population are transformed by three genetic operators, namely *selection*, *crossover* and *mutation*.

- Selection

  Often called *reproduction*, it is a process that probabilistically selects an individual i to remain in the population and reproduce with probability $p_i = f_i / \sum_{j=1}^{N_{POP}} f_j$, where N_{POP} is the number of individuals in a population. The fitness of each individual is set in the evaluation stage. The effect of selection is that individuals having above-average performance reproduce while those poorly performed individuals are culled from the population.

- Crossover

  Crossover is a process where a substring of a chromosome is exchanged with the corresponding substring of its mating partner. New structures are generated in this process with the hope for better average fitness. The length of the string to be swapped is selected randomly. Crossover results in a randomized yet structured information exchange. Each new individual generated combines the characteristic of both parents.

- Mutation

In order to avoid the search process being stuck in local optima, it is necessary to introduce some "noise" to the system. Mutation is employed for such a purpose. It randomly flips a bit in a chromosome's string representation. The resulting effect prevents the search from premature convergence. The probability of mutation should not be too large, otherwise good gene structures will be lost, which in turn will delay the search process.

Fig. 2.6 illustrates a flowchart of a GA.

GAs have been used widely for the task scheduling problem [6, 25, 50, 95, 101, 105, 110]. In the following, we briefly discuss some related work.

Hou et al.'s algorithm

Hou et al. [50] introduced a genetic algorithm (we will call it HAR algorithm hereafter) for DAG scheduling. It is one of the earliest attempts to use GA to solve this type of scheduling problem. In this algorithm, each individual s is composed of m strings $\{s_1, s_2, ..., s_m\}$. Each string contains the tasks scheduled to a processor represented by the string. These tasks appear in the order of their execution in the schedule s. It is possible that some strings may represent schedules not satisfying the precedence constraints (infeasible). The authors proposed a method to guarantee that all strings in the initial population and consequently generated population are feasible solutions to the problem.

Let $PRED(T_i)$ denote the immediate predecessors of task T_i and $SUCC(T_i)$ the

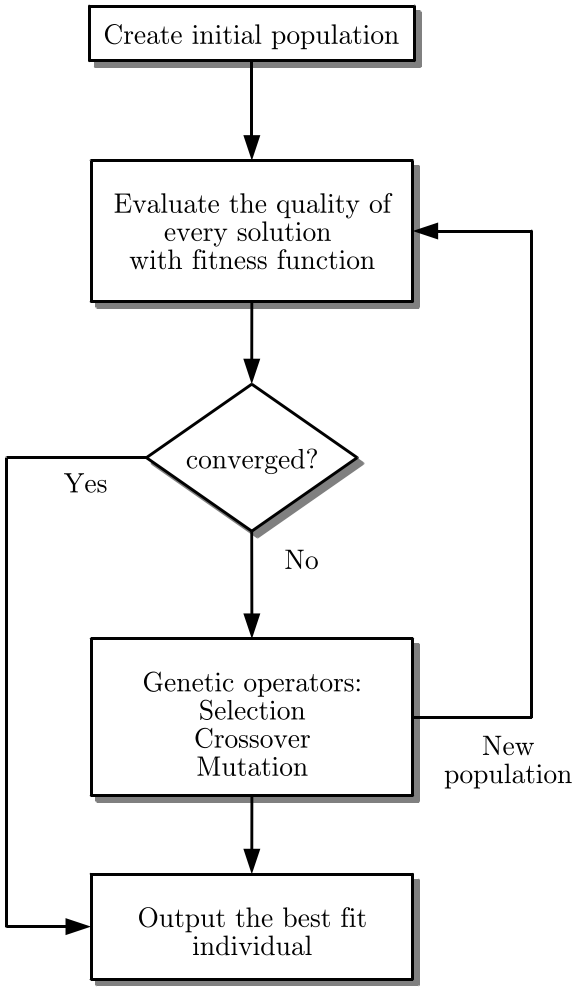


Figure 2.6: An outline of genetic algorithms

immediate successors of T_i. The height of a task T, $height(T)$, is defined as the maximum length between T and an exit node. Each task T_i is assigned a random height between $\max\{height(T_j)\} + 1$ and $\min\{height(T_k)\} - 1$, over all $T_j \in PRED(T_i)$ and $T_k \in SUCC(T_i)$. Tasks in each string are ordered according to their heights in order to guarantee the feasibility of a given schedule coded in the above fashion.

The initial population is randomly generated. First the height of every task is computed. For each height h, choose r tasks at random to be assigned to p_1. Repeat this step for all processors except the last one. Finally assign all remaining tasks to the last processor.

Selection is based on the "roulette wheel" principle where each string in the population occupies a slot size proportional to its fitness value. Then random numbers are generated and used as indices into the wheel to select strings to be passed for crossover and mutation. The better the fitness of an individual, the greater the chance of it being selected.

During the crossover, each string of the two parents is cut into two halves - left and right. This is done as follows: first a random height h is chosen. Tasks whose height are larger than h become the right half and those with height smaller than h form the left half. Only the right halves of the strings are exchanged. To guarantee the feasibility of generated strings, the crossover site must meet two conditions: (1) the height of the tasks next to the crossover sites are different, (2) the height of all the tasks immediately in front of the crossover sites are the same. Mutation is implemented by randomly

exchanging two tasks with the same height.

Corrêa et al. [25] pointed out that there are a few shortfalls existing in the above algorithm. First, in the initial population, the tasks are not uniformly distributed over all the processors due to the initial population generation scheme. Secondly, the crossover operator cannot generate some feasible solutions. Finally, there is no knowledge about the problem being integrated into the algorithm. Based on these observations, they proposed a new algorithm.

Corrêa et al.'s algorithms

Corrêa et al. [25] designed two improved algorithms based on HAR algorithm. The *Full Search Genetic Algorithm* (FSG) uses the same encoding mechanism as HAR. However, it creates the initial population differently. Specifically, each individual is generated through a random list heuristics. Ready tasks are selected randomly and assigned on random processors. This scheme ensures that tasks are uniformly distributed over the processors. Selection is implemented using a biased roulette wheel as in HAR.

In order to avoid omitting some feasible solutions during the crossover, a new crossover scheme is designed. Given a schedule s, the disjunctive graph of G is defined as $\mathcal{D}(s) = (V, E(s))$, where $E(s) = E \cup \{(n_i, n_j) | n_i$ and n_j are assigned to the same processor in that order$\}$. The task set V is a *closed task set* if all the predecessors of any task from V also belong to V. The first step of crossover is to divide each of the two individuals s_1 and s_2 into two parts. In order to guarantee consistency, the left part of a partition (V_1 and V_2) must be a closed task set. Let $G' = (V, E(s_1) \cup E(s_2))$

and $V' = V$. The partition is carried out as follows:

(1) Randomly select a task n_i from V. Follow the two rules to insert related task nodes to left (V_1) or right (V_2) part.

- n_i and all of its predecessors that remain in V' are inserted in V_1
- n_i and all of its successors that remain in V' are inserted in V_2

(2) Remove all tasks inserted from V'.

(3) Repeat steps from (1)-(2) until V' becomes empty.

The two offspring s'_1 and s'_2 are generated as in HAR. For the mutation of individual s, first the disjunctive graph $\mathcal{D}(s) = (V, E(s))$ is constructed. The new individual is generated using the method for generating each individual of the initial population.

A combined genetic-list algorithm (CGL) is also proposed. The salient feature of this algorithm is the combination of list scheduling algorithm with crossover and mutation operators. In the crossover operator, instead of just exchanging the right parts of the involved strings, additional knowledge is introduced. The tasks in the right part of the strings are scheduled according to a list algorithm over $\mathcal{D}(s_2)$. Any list algorithm should work here. The authors chose to use *earliest data/most immediate successor first* (ED/MISF) algorithm. Similarly, during the mutation of s, the new individual is formed by a list heuristic.

Wang et al.'s algorithm

Wang et al. [101] designed a GA based scheduling method, which has the following characteristics. The encoding scheme uses two strings to represent an individual schedule: the *matching string* and the *scheduling string*. The matching strings contain information about where each task is assigned. The scheduling string is a topological sort of the DAG and is used by the evaluation step. In generating the initial population, along with the randomly generated schedules, an individual that represents the solution from a non-evolutionary heuristic is also included. This may reduce the time needed for convergence. Two roulette wheel selection schemes are employed in the selection step. The rank-based scheme assigns the angle of the sector allocated for an individual based on its rank. In the value-based scheme, the angle is proportional to f_{avg}/f_i, where f_i is the fitness of individual i and f_{avg} is the average fitness. In addition, the best individual is guaranteed to be passed onto the next generation. This is called *elitism*.

The crossover operator deals with the two strings of an individual separately. For the scheduling strings, it randomly generates a crossover point on each string of the mating pair. It divides the strings into two parts - top and bottom. The top parts remain the same, and the tasks in the bottom parts are reordered. The new ordering of the tasks in one bottom part is the relative positions of these tasks in the other original scheduling string of the mating pair. For the matching strings, the crossover operator randomly chooses a point and exchanges the bottom parts of each string.

The mutation operator handles the two strings differently. For the scheduling string,

the operator selects a task at random. It randomly move the task to another position in *valid range*. The valid range of a task is the set of positions in the scheduling string where the task can be scheduled without violating the precedence constraints. For the matching string, the operator randomly selects a task/machine pair. The machine assigned to the task is changed randomly to another machine.

Wu et al.'s algorithm

An incremental GA for DAG scheduling has been proposed recently [105]. This algorithm has two distinct features. First, each individual in a GA population is composed of a certain number of cells. A *cell* is actually a task and processor pair - (t, p), which states that task t is a assigned on processor p. The order in which the cells appear on an encoding dictates the order in which the tasks will be executed on each processor. One significant difference from the previous GAs is that task duplication is allowed. The same task may be assigned to different processors. The second and most important feature of this GA is the design of the fitness function. The fitness of an individual is a weighted sum of two partial fitness values. The first part, *task fitness*, measures how close the individual is to be a valid schedule. By incorporating this factor, the need to ensure a valid schedule is not necessary any more, and partial solutions can be used to direct the search. The second part, *processor fitness*, evaluates the quality of the schedule in terms of the makespan. The overall fitness of an individual is calculated by:

$$fitness = (1 - b) \times task\_fitness + b \times processor\_fitness \tag{2.3}$$

where $b \in [0.0, 1.0]$ is a user controlled parameter. By adjusting the value of b, the algorithm can encourage either the formation of valid solutions or the formation of optimal solutions. Simulation shows that both components of the fitness value are necessary for the evolution of optimal solutions.

2.3 Static DAG scheduling in non-deterministic HDCS

In the previous section, we surveyed some representative algorithms for offline scheduling DAG type applications in a deterministic environment, where it is assumed that the prediction of task execution time and network bandwidth/latency is accurate. However, in a real world computing environment, the probability of a precomputed schedule being executed exactly as expected is low. Due to resource sharing among multiple users, the performance of resources is inherently variant.

The dynamic characteristic of a real world computing environment could severely penalize the performance of schedules obtained based on the predicted resource characteristics. An optimal schedule based on inaccurate expectations about the real environment will considerably deviate from optimal when executed. Kidd et al. [58] noted that the optimization criteria commonly used to minimize the makespan, based on the expected run-times of each job on each machine, is incorrect because it gives an average schedule completion time that is always underestimated. Schedules generated using such incorrect criteria will result in poor performance if carried out in a real environment. The authors pointed out that using both the expected execution times and their dis-

tribution information will lead to better schedules. Consequently, in an uncertain, real world computing environment, it is not desirable to devote significant effort to finding an "optimal" schedule because the real optimal schedule can not be obtained without its execution in a real environment. On the other hand, a schedule that might not be very close to optimal before execution, but contains built-in mechanisms for handling the uncertainty of real environment, may turn out to be a well-performed schedule upon execution.

2.3.1 Classifications of current research on scheduling with uncertainties

Davenport et al. [30] classified the approaches to dealing with uncertainty in a scheduling environment into two categories; *proactive* and *reactive* scheduling. The goal of proactive scheduling is to build schedules that contains some flexibility to hedge against uncertainty. In other words, by considering the uncertainty information when producing a schedule, it aims to generate a schedule that is more *robust*. The authors define a robust schedule as one that is likely to remain valid under different disturbances and one where violation of the assumptions upon which it is built are of no or little consequence. It must also have the ability to satisfy performance requirements predictably in an uncertain environment. Although defined in the context of *Job Shop Scheduling* and *Resource Constrained Project Scheduling*, this definition is applicable to the scheduling problems we consider. In reactive scheduling, a schedule is not computed before the actual execution begins. The scheduler decides when and where to execute a task based

on the current information about the status of the system and perhaps an existing preliminary schedule. Therefore, a schedule is revised or modified as necessary when the status of the system changes. Based on the degree of which the schedule modifies a predictive schedule, several approaches can be followed. In one scenario, a schedule decision is made dynamically in order to take into account the variance of environment characteristics. As a result, task dispatching is done at run time. In this case, no predictive schedule is necessary. At the other end of the spectrum, a completely new schedule is generated every time a schedule is not executed as planned. This requires large amounts of computation time to produce new schedules. In between the two extremes, a scheduler might repair the existing predictive schedule to account for the current state of the system. This balances the scheduling complexity and the incorporation of up-to-date system state. The authors also noted that a practical scheduling algorithm is very likely to employ both proactive and reactive techniques. The combination of the two types of techniques can results in a schedule with quick response time and good quality under the uncertain environments.

Polices [79] categorized the different approaches for scheduling with uncertainty into the following 4 groups:

(1) Robust scheduling: algorithms in this group use the information about possible uncertainty of the system to produce schedules capable of "absorbing" some amount of uncertainty.

(2) Partially defined schedule: algorithms in the category define a partial order of

scheduling tasks and use such flexibility to protect the schedule from uncertainty. No information about uncertainty is used.

(3) Rescheduling: A schedule is modified dynamically if the uncertainty causes a certain degree of deviation from the expected performance. One possibility is to repair as local as possible to maintain the stability of the schedules. Another approach is a global repair to produce better quality schedules.

(4) Dynamic scheduling: there is no baseline schedule required. The next task to be executed is selected and scheduled based on the current status of the system.

The first two groups fall into the category of proactive scheduling and the last two belong to reactive scheduling as defined by Davenport et al. [30].

In the context of task scheduling in HDCS such as the grid, Bölöni et al. [16] suggested several possible approaches for scheduling with non-deterministic task execution times:

(1) Overestimate the execution time of each task to minimize the probability of exceeding the allocated time slot on each processor. This will certainty reduce the utilization of the resource.

(2) Compute the schedule at run time dynamically. Once the task node is ready to be executed, information about the current state of the system is gathered to produce a new schedule for the remaining tasks of the tasks graph.

(3) Precompute a set of schedules for various scenarios of the system. At run time, choose the schedule upon which the condition best fits the current condition is built.

(4) Find schedules less vulnerable to uncertainty, i.e. more robust, statically.

Based on the classification criteria of Davenport et al. [30], approaches (1), (3) and (4) are proactive scheduling algorithms; approach (2) is reactive scheduling algorithm.

In the following, we describe different approaches to scheduling with uncertainty. We use the classification of Davenport et al. [30] in our description.

2.3.2 Review of different scheduling techniques

Proactive scheduling

Robust task scheduling One of the most important approaches in the category of proactive scheduling is robust task scheduling. A robust schedule should be able to tolerate some degree of uncertainty in the execution environment. Although robustness is an important metric of the quality of a schedule, little work has been done on how to generate robust schedule, especially in the area of DAG scheduling.

In [27], the authors studied the problem of scheduling n independent jobs on a single machine. The processing times of individual jobs are uncertain and can be described through a set of processing time scenarios Λ. Two versions of robust scheduling problems are proposed. The *Absolute Deviation Robust Scheduling Problem* (ADRSP) aims to find the schedule that minimizes the worst-case absolute deviation from optimality for

the total flow time. It is formulated as:

$$\min_{X}\{\max_{\lambda\in\Lambda}[\varphi(X,P^{\lambda})-G^{\lambda}]\} \tag{2.4}$$

where $\lambda \in \Lambda$ represents a set of processing times of the tasks that can be realized with some probability. The vector $P^{\lambda} = \{p_i^{\lambda} : i = 1, 2, ..., n\}$ denotes the processing time of each job corresponding to scenario λ. Let σ_{λ}^{*} be the optimal sequence given processing time scenario λ and G^{λ} is total flow time of sequence σ_{λ}^{*} using some heuristic such as the shortest processing time (SPT) schedule. In the formulation, $\varphi(X, P^{\lambda})$ is the total flow time of scheduling X given processing time scenario λ. Another similar version is called the *Relative Deviation Robust Scheduling Problem* (RDRSP), which determines the schedule that minimizes the worst-case percentage deviation from optimal for the total flow time. It can be formulated as:

$$\min_{X}\{\max_{\lambda\in\Lambda}[\varphi(X,P^{\lambda})/G^{\lambda}]\} \tag{2.5}$$

After proving the NP-hardness of the problems, the authors gave a branch-and-bound algorithm for the problem and two heuristic approaches for the two-job absolute and relative robust scheduling problems.

Another example of considering robust job shop scheduling is found in [66]. The uncertainty in the job shop scheduling problem is typically caused by machine breakdown and subsequent disruption to the schedule. Let s be a schedule that specifies the order

in which tasks are executed on machines, $M_0(s)$ the makespan of s without disruptions and $M(s)$ the actual makespan in the presence of disruptions. Thus, the schedule delay is $\delta(s) = M(s) - M_0(s)$. Let r be a real value in the interval $[0, 1]$. The robustness of schedule s, $R(s)$, is defined as:

$$R(s) = r \times E[M(s)] + (1 - r) \times E[\delta(s)] \tag{2.6}$$

where $E[\cdot]$ represents the expectation operator. When there is only one disruption, $R(s)$ can be computed analytically. If more than one disruption is considered, the problem becomes intractable. The authors developed several surrogate robustness measures that are strongly related to the expected delay and makespan to solve the problem. Among those measures, $RM3(s)$ is shown to be the most effective. It is defined as:

$$RD3(s) = \frac{\sum_{i \in N_f} slack_i}{|N_f|} \tag{2.7}$$

and the associated expected makespan is

$$RM3(s) = M_0(s) - \frac{\sum_{i \in N_f} slack_i}{|N_f|} \tag{2.8}$$

where $slack_i = lst_i - est_i$ is the difference between task i's latest start time (lst) and earliest start time (est). N_f is the set of tasks to be processed on fallible machines. A genetic algorithm is designed to generate schedules based on the robustness measure

defined. In the GA, the fitness of a schedule s, $fitness(s)$, is computed from $RD3(s)$ and $RM3(s)$ as follows:

$$fitness(s) = \frac{\{MAXZ - Z(s)\}^{FS}}{\sum_{i \in PS} \{MAXZ - Z(i)\}^{FS}} \tag{2.9}$$

where $MAXZ$ is a sufficiently large number and $Z(s)$ is the objective function defined as:

$$Z(s) = r \times RM3(s) + (1 - r) \times RD3(s) \tag{2.10}$$

PS is the population size and FS is the fitness selection power. Experiment results show that when r is close to 1, the algorithm is able to find robust schedules that maintain good performance in the expected makespan and reduce the expected delay. When the expected delay is the objective ($r = 0$), the algorithm finds the schedule with the best expected delay performance but it comes at the expense of the makespan.

Darbha et al. [29] investigated the impact of change in computation and communication costs on the precomputed schedules. Here, the variations is due to the inaccuracies of estimating the instruction execution times or the message passing delays. Given a DAG $G(V, E, \tau, c)$, let s denote the schedule generated for the DAG using a heuristic developed by the authors called STDS. Let $L(G(V, E, \tau, c), s)$ represent the makespan of s under the cost (τ, c). The schedule s may or may not be generated using (τ, c). The

robustness of scheduling algorithm A, $R(A)$ is defined as:

$$R(A) = \frac{L(G(V, E, \tau_2, c_2), s_2)}{L(G(V, E, \tau_2, c_2), s_1)} \tag{2.11}$$

Here, schedule s_1 is obtained with the estimated cost (τ_1, c_1) and s_2 is generated with real costs (τ_2, c_2). Therefore, $L(G(V, E, \tau_2, c_2), s_2)$ represents the makespan that would be obtained by applying the real costs (τ_2, c_2) with schedule s_2. Similarly, $L(G(V, E, \tau_2, c_2), s_1)$ is the makespan of s_1 with real costs (τ_2, c_2). The robustness of the algorithm is the ratio of the two makespans. The desired robustness is a value very close to 1.0. Based on this definition, the authors identified some conditions for their heuristic STDS to be robust and showed that STDS is robust through experiments. The authors pointed out that the robustness of the resulting schedule can be an important objective for scheduling strategies rather than just attempting to minimize the makespan.

An intuitive notion of robustness of a schedule is that the execution of the schedule should be able to maintain performance despite various uncertainties in the resource environment. Although intuitive, it is difficult to measure it quantitively. There is no consensus on which definition should be used. Different researchers propose their own definition of robustness. For example, in [7] the authors presented a metric for the robustness of a resource allocation. Several steps need to be followed in their procedure. First, all the *robustness requirement* that make the system robust are selected quantitively. Next, *perturbation parameters* whose values may impact the quality of

performance features selected in the previous step are identified. Then, the perturbation parameters' impact on the system performance are evaluated by mathematically describing the relationship. Finally, the robustness of the system is the smallest variation in the perturbation parameters that cause the degradation of the system to violate the performance bounds.

Along this line, England et al. [37] proposed another new metric to measure the robustness. It essentially uses the probability hypothesis theory to measure how close two probability distribution are. One of the distributions $F(x) = P(X < x)$ is the cumulative distribution function (CDF) of performance under normal operating conditions. The other one, $F^*(x) = P(X < x)$ represents the CDF of performance with the presence of perturbation. A robust system should have very similar $F(x)$ and $F^*(x)$. To measure the closeness of these two CDFs, a Kolmogorov-Smirnov (K-S) test is performed. It calculates the δ as defined by:

$$\delta = \sup_{-\infty<x<\infty} F(x) - F^*(x) \tag{2.12}$$

where δ can be used to measure the amount of performance degradation.

Bölöni et al. [16] investigated the problem of robust scheduling of a DAG modeled task graph. A surrogate measurement of robustness of a schedule is proposed based on the concepts of *spare time* and *slack*. Consider the task graph $G = (V, E)$. An augment graph G' is first constructed by adding host dependencies to the original graph. If two tasks are scheduled sequentially on the same host, a new host dependency link is added

between them. If a data dependency link between the two tasks is already in place, no new link will be added. Consider an edge e_{ij} in G', the spare time of e_{ij} is defined as:

$$t_{spare} = t_{s_j} - t_{f_i} \tag{2.13}$$

where t_{s_j} is the start time of task j on its allocated processor and t_{f_i} is the finish time of the task i. The slack of task i, σ_i, is the minimum spare time on any path from i to an exit task. A *critical* task has zero slack. A *safe* task has slack larger than the upper bound delay of the task. The authors provide an empirical formula for computing the robustness of schedule s:

$$R(s) = \sum_i \frac{\min(\sigma_i, t_i^{upper} - t_i)}{t_i^{\gamma}} \tag{2.14}$$

where t_i is the estimated execution time of task i, and t_i^{upper} is the upper bound of that execution time. $\gamma \in (0, 1]$ is a value determined experimentally. Experimental results show that robustness analysis can increase the number of safe tasks and improve the robustness of the resulting schedule.

Stochastic scheduling Stochastic scheduling approaches assume that the resource characteristics can be modeled with random variables with certain probabilistic distributions. The information about uncertainty is known *a priori*. The idea of using stochastic information during the schedule has been addressed in several studies [34, 43, 54, 58].

In [54], the impact of predictive inaccuracies on job selection and resource allocation

is investigated. The authors claimed that if a resource allocation policy is based on the expected execution time of the job on different resources, then it might cause the schedule to make a bad scheduling decision.

The same observation is made in [58]. The authors pointed out that scheduling algorithms that generate schedules based on the expected run-times of each job on each machine will result in an inferior schedule if the assumption that actual run-times are exactly the same as the expected ones is violated. Consider the problem of scheduling J independent tasks onto M machines. e_{jm} is the random variable modeling the execution time of task j on machine m. The expected value of e_{jm} is $E(e_{jm})$. Let t_m denote the time when machine m finishes all the tasks assigned on it. That is,

$$t_m = \sum_{j \in \mathcal{U}_m} e_{jm} \tag{2.15}$$

where $\mathcal{U}_m$ is the set of tasks assigned on m. Two versions of optimization criteria can be used when finding the schedule. The first one is

$$\min \hat{z} \tag{2.16}$$

where

$$\hat{z} = \max_{m=1..M} \hat{t}_m \tag{2.17}$$

and t_m is defined above. Here, e_{jm} is the random variable. The other criteria is:

$$\min \hat{z} \tag{2.18}$$

where

$$\hat{z} = \max_{m=1..M} \hat{t}_m \tag{2.19}$$

and

$$t_m = \sum_{j \in \mathcal{U}_m} E(e_{jm}) \tag{2.20}$$

It is proved that makespan calculated based on the expected run-times is less than or equal to the actual schedule completion time. Another observation is that by using the second optimization criteria, even the "optimal" schedule obtained via an exhaustive search can be a poorly performed one in the real resource environment where the actual execution times of the tasks are different than the expectations. Based on these observations, the authors concluded that scheduling algorithms that use not only the expected execution times but also their distributions can obtain better schedules.

Fujita et al. [43] also investigated the problem of scheduling independent tasks with inaccurate estimation of execution cost. The *Robust Multiprocessor Scheduling Problem* (RMSP) is such that, given m identical processors and n independent tasks with estimated execution costs and inaccuracy distributions, generate a schedule such that the expected makespan of the overall schedule will be minimized. A heuristic for tolerating the inaccuracy estimation is proposed. Basically it is to solve the set partition problem

in a greedy fashion.

Dogan et al. [34] used genetic algorithm to solve the same problem. The goal is to produce better schedules in terms of minimizing the makespan under the real resource environment where the actual task execution times are not the same as the expected ones. Task execution times are modeled as random variables. Two similar observations as those in [58] are made: (1) Makespan of a schedule is calculated based on the expected execution times will underestimate the expectation of the makespans obtained with the real execution times. (2) Even an optimal scheduling obtained by only using the expected execution times could be far from the actual optimal solution. Most heuristics consider the following problem:

$$\min_{\mathcal{X}\in\pi}\{\hat{T}_F(\mathcal{X})\} \tag{2.21}$$

where

$$\hat{T}_F(\mathcal{X}) = \max_{m_j\in M}\{\sum_{v_i\in V} x_{i,j}\bar{\tau}_{i,j}\} \tag{2.22}$$

Let $\mathcal{X}$ be a schedule, then the makespan under $\mathcal{X}$ is

$$T_F(\mathcal{X}) = \max_{m_j\in M}\{\sum_{v_i\in V} x_{i,j}\tau_{i,j}\} \tag{2.23}$$

where M is the set of processors and V is the set of tasks to be scheduled. $x_{i,j}$ is the assignment scheme: if $x_{i,j} = 1$ if and only if task v_i is assigned to processor m_j. Otherwise, $x_{i,j} = 0$. Let $\bar{T}_F(\mathcal{X}) = E[T_F(\mathcal{X})]$, so the other version of the problem can

be formulated as:

$$\min_{\mathcal{X}\in\pi} \bar{T}_F(\mathcal{X}) \tag{2.24}$$

where π is the set of all valid task assignments. To solve this problem, the authors first computed $\bar{T}_F(\mathcal{X})$ based on probability theory. Then three methods are used to generate schedules, a popular heuristic called RC that ignores the variances of execution times, a genetic algorithm with objective function $T_F(\mathcal{X})$,and a genetic algorithm with objective function $\hat{T}_F(\mathcal{X})$. Experiments show that GA using $T_F(\mathcal{X})$ as its objective function accounts for the stochastic nature of the task execution times and thus is able to find schedule with better makespan when executed in the real resource environment. In addition, this algorithm is able to decrease the standard deviation of the makespan. The limitation of this genetic algorithm is that it is not always possible to calculate $\bar{T}_F(\mathcal{X})$ analytically where each execution time can assume a different probability distribution. Even though each execution time has the same probability distribution, only certain types of distribution can lead to a mathematically tractable solution.

Stochastic information about the execution times is also used in [56] to more accurately estimate the EST of a task node. Two new versions of ETF and DLS algorithms are developed where the new method of calculating EST based on the mean and variance of each task's execution time is employed. Results show that the stochastic versions of the two algorithms can find schedules that have smaller average makespans than those found by their static counterparts. Furthermore, the stochastic EFT and DLS are able to predict the actual makespan of a generated schedule more accurately. However, the

approach taken to calculate the EST limits itself to the case where the execution time of each task is uniformly-distributed.

Stochastic scheduling addresses the problem of incorporating stochastic information about the resource environment and the tasks while making scheduling decisions in order to generate a better schedule. Random features such as task execution times and network transfer rate are modeled by specifying their probability distributions, which are assumed to be known by the scheduler. Although there are a few studies on this subject, it seems that there is no consensus on how to efficiently use the stochastic information and how to evaluate the generated schedule.

Reactive scheduling

Reactive scheduling algorithms make scheduling decisions based on the current state of the system at run-time. Sometimes they are also referred to as *online scheduling* or *dynamic scheduling*. It is common that a reactive scheduling algorithm also uses a pre-computed schedule that is generated based on the estimations of resource performance, i.e., it is a combination of offline and online scheduling techniques. In the following, we briefly describe some reactive scheduling algorithms reported in the literature.

In [70], a dynamic scheduling algorithm called *hybrid remapper* was developed. As the name suggested, it is a hybrid of a static and dynamic scheduling algorithm. The algorithm is composed of two phases. During the first phase, an initial schedule based on the estimations of task execution times is generated. In the second phase, while the application is executing, the remapper uses run-time values for the task completion

and machine available times whenever possible to modify the initial schedule in order to reduce the final makespan. Three variants of the hybrid remapper algorithm are described. Each variant has a common first step, where tasks are partitioned into blocks and assigned ranks. The rank is an indication of the priority for the task to be mapped. These variants differ in the minimization criteria they use and in the way they order the tasks examined by the partial mapping problem during the second step. One of the variants seeks to minimize the expected partial completion time at each remapping step, and the others attempt to minimize the overall expected completion time. Two variants of the hybrid remapper order the subtasks at each remapping step using ranks computed at compile time, and the other uses a parameter computed at run time. Although differing in certain aspects, the remapper aims to improve the initial schedule by using both the run-time information that becomes available during application execution and the information that was obtained prior to the execution of the application.

Alhusaini et al. [4] took a similar approach for scheduling independent task on the Grid [41]. The objective is to minimize the overall makespan of all tasks while satisfying all resource sharing constraints among them. A two-phase scheduling algorithm is proposed. The first phase is an offline planning phase where a preliminary schedule is generated at compile-time. The second phase is a run-time adaptation phase where run-time information, such as the variation in computation and communication costs and the early release of resources, are taken into account in order to improve the perfor-

mance of the preliminary schedule. By comparing the algorithm to a baseline algorithm with no adaptation at run-time, the authors showed that the new algorithm can significantly improve the makespan. It also outperforms a dynamic algorithm that does not use a preliminary schedule.

In [74, 75], the problem of scheduling a DAG-modeled task graph onto a system with online communication disturbance is studied. In this model, an estimation of the communication cost is known at compile time. However, due to network contention, link failure etc., the actual cost is disturbed at run-time. The authors claim that because of the lack of exact communication cost, building a full-fledged schedule at compile time is inappropriate. On the other hand, building the schedule completely at run-time is also unsatisfactory. Therefore, a trade-off between these two approaches is proposed. This approach has three steps:

(1) Compute an offline schedule based on the estimated communication cost.

(2) Compute a partial order that includes both the original task dependence $\prec$ and the new machine dependence where if i and j are assigned to the same processor in that order, then $i \prec_p j$.

(3) At execution time, use the ETF policy to get a complete schedule including assignment of step (1) and partial order from step (2).

Simulation results show that this algorithm performs better than the one with schedule fully generated at run-time.

Boyer et al. [17] investigated the problem of dynamically scheduling a task graph with inaccurate task execution time estimation. Two algorithms are proposed. The first one, called *Dynamic Adaptive Random Scheduler*, is a hybrid approach. It searches a near optimal schedule based on the current estimated task execution time, then executes the schedule. Later, task migration is allowed based on the actual execution time. The *Load Balancing* (LB) algorithm initially distributes tasks via a heuristic that uses estimated task execution times and accounts for heterogeneity and dependencies. While the tasks are executing, they are migrated to rebalance the load whenever a processor becomes idle in a way that the number of migration is minimized. In addition, task dependence constraints are not violated. Three rules must be followed during a migration. (1) The tasks selected from migration are always taken from the tail of the queue of the overloaded processor. (2) When multiple tasks are chosen to migrate, their relative execution order on the new processor is kept the same as the one in the old processor. (3) When a task is migrated to a new processor, it is merged into the queue of the target machine in a way that the queue remains topologically sorted.

Recently, scheduling with uncertainty has been studied in the context of the Grid [14, 73, 86, 87, 108]. In [108], the authors classified dynamic scheduling in the uncertain Grid environment into two categories; *prediction-based* and *just in-time* scheduling.

The prediction-based, dynamic scheduling uses dynamic information as well as some initial schedule based on prediction. The scheduler is required to predict the performance of task execution on resources and to produce a near optimal schedule for the

task at compile time. During the execution of the initial schedule, the schedule is changed dynamically based on the current information about the Grid. For example, in the GrADS framework [14], the scheduler first generates an initial schedule by using the prediction on the task execution time with application performance models and network transfer rate with MDS [40] and NWS [104]. In the mean time, an initial contract that specifies the expected performance of tasks on the assigned resources is created. Due to the uncertainty of the Grid environment, this contract will be probably violated. The task will be migrated to other resources if this happens. Another cause of migration is that a better resource for the task becomes available. This is called *opportunistic migration*. In [86], a low-cost rescheduling policy for the mapping of workflows on Grids was developed. An initial schedule is first built based on estimates. The rescheduling policy then evaluates, at run-time before each task starts execution, the starting time of each node against its estimated starting time in the static schedule and the *slack*, in order to make a decision for rescheduling. The slack of each task is an indication of the maximal value that can be added to the execution time of this task without affecting the overall makespan of the schedule. The algorithm will proceed to a rescheduling action if any delay between the real and the expected start time of the task is greater than the slack. The purpose of such a policy is to optimize the makespan of the schedule while minimizing the frequency of rescheduling attempts. Prediction-based scheduling is also used in [87]. An initial contract between the client and service provider is created with the estimates of job completion time. This contract is monitored throughout the

course of execution. Once it is violated, dynamic repair and rescheduling operations are triggered. A metric called *surety* is proposed. The surety is the probability that a task will finish its execution within a deadline window predicted by the service provider. During the monitoring phase, surety is updated whenever progress is made or when a certain period of time has passed. If the surety drops below the minimum threshold determined by the contract, the scheduler will contact all service providers to get new bids. Then a bid that can finish the task at the earliest time is accepted.

Just in-time scheduling only makes scheduling decisions at the time of task execution. Due to the inherently uncertain and dynamic resource performance and availability of the Grid environment, scheduling before run-time could result in a poor schedule. Another factor that makes static scheduling unreliable is that accurately predicting the execution time of all task on a shared Grid resource is extremely difficult. In [33], a just in-time planning scheme of workflow application is described. The original workflow is first partitioned according to a specified partitioning algorithm. This creates a new workflow where the nodes are partial workflows. Then, the Pegasus system maps and submits the partial workflows to the Grid. Dependency between two partial workflows are preserved by enforcing that the mapping of a dependent workflow is not started until the preceding workflow has finished executing. DAGMan [97] is used to control the in-time planning process by making sure that Pegasus does not refine a partial workflow until the previous partial workflow successfully finished execution.

Chapter 3

Task Scheduling Considering Different Processor Capabilities

3.1 Introduction

As described in Chapter 2, the scheduling of task graphs is highly critical to the performance of heterogeneous distributed computing system. It deals with the allocation of individual tasks to suitable processors and proper order assignment proper order of task execution on each resource where the common objective is to minimize the overall completion time or *makespan* [15, 36, 64, 65]. As the DAG scheduling problem is *NP-complete* in general, a number of heuristics have been proposed. List scheduling based heuristics usually generate good quality schedules at a reasonable cost. Various methods to specify the priorities of nodes and select the best processor have been pro-

posed [65, 94, 98]. List scheduling heuristics are originally designed for homogeneous systems where processor speed and capability, network bandwidth between any pair of processors are the same. It has been extended in two directions. Firstly, several *dynamic* list scheduling algorithms have been introduced [63, 94, 107]. These algorithms update the priorities of each node and the scheduling list dynamically at each step. Similar to traditional list scheduling algorithms, at each step the node with highest priority is selected for scheduling. Dynamic list scheduling can potentially generate better schedules. However, these approaches can significantly increase the time complexity of the algorithms. Secondly, a number of new list scheduling algorithms for heterogeneous environment have been proposed [82, 94, 98]. A comparison of those algorithms reveals that during the processor selection phase: (1) insertion-based policy, which allows the possible insertion of a task in an earliest idle time slot between two already-scheduled tasks on a processor, is better than non-insertion based counterparts. (2) processor selection criteria that consider the different processor speed (e.g., *Earliest Finish Time*) outperform those do not include this factor (e.g., *Earliest Start Time*).

Although the DAG scheduling in general is a well studied problem, most of the algorithms assume that the processors are equally capable, i.e. each processor can execute *all* the tasks with possibly different speeds. While some of the algorithms do not make the assumption explicitly, they also do not consider the potential effect of different capabilities either [69, 85, 98]. Thus, these algorithms suffer in performance when scheduling under this situation. Other algorithms simply become inapplicable

without modification. For example, the Critical-Path-on-a-Processor (CPOP) algorithm introduced in [98] allocates all critical tasks onto a single processor in an attempt to minimize the total execution time of the critical tasks. This algorithm fails if none of the processors can process all the critical tasks. Another category of algorithms which becomes unsuitable is the clustering algorithms [59, 68, 106, 107]. An algorithm of this type allocates tasks into different clusters. Each cluster can contain more than two tasks. When two tasks are assigned to the same cluster, they are executed in the same processor. Under the condition of processors with different capabilities, chances are none of processors can carry out all the potentially large number of tasks in the same cluster. Therefore, unless effectively modified, clustering algorithms can not be directly used under these circumstances.

In this chapter, we propose a new static list Scheduling algorithm for heterogeneous processors with Different Capabilities (SDC). As found in [109], the methods used to assign weights to the nodes significantly affect the performance of scheduling algorithms. We suggest a new approach of setting a task node's weight. It considers the percentage of capable processors as well as the task's average execution cost among those capable processors. The SDC algorithm selects the task with the highest *b-level* [65] at each step. The selected task is then assigned to a processor that minimizes its *Adjusted Earliest Finish Time (AEFT)* (defined in Section 3.3) with an insertion-based policy. The AEFT adapts the EFT by including a new term that indicates how large the communication between the current node and its children will be on the average, provided that

it is scheduled on the current processor. Due to resource scarcity, processors that minimize EFT for the current scheduling node are not necessarily the best choice because of potentially overwhelming inter-processor communication between the node and its children as shown by the example in Section 3.3. The algorithm has been tested on a large number of randomly generated problems of different sizes and types. The parametric graph generator is similar to the one designed in [98] but with a different set of parameters. We compare SDC with two other list scheduling algorithms, the Heterogeneous Earliest Finish Time (HEFT) [98] and Dynamic Level Scheduling (DLS)[94]. *Normalized Schedule Length*(NSL) and *Average Percentage Degradation*(APD) [64] are used as the comparison metrics. The comparison study shows that our algorithm performs considerably better in most cases, especially when the *Communication-to-Computation Ratio* (CCR) and *Percentage of Incapable Processor* (PIP) are large.

In the next section, the scheduling problem and some related terminology are defined. Our algorithm (SDC) is introduced in Section 3.3. Section 3.4 presents experimental results based on randomly generated task graphs and a real world bioinformatics workflow application graph. Section 3.5 contains the concluding remarks.

3.2 Problem description

A scheduling system usually consists of three parts; application, computing environment, and scheduling goal. The application and computing environment can be represented by a task graph and resource graph respectively.

3.2.1 Task graph

The DAG is a generic model of a workflow application consisting of a set of tasks (nodes) among which precedence constraints exist. It is represented by $G = (V, E)$, where V is the set of v tasks that can be executed on a subset of the available processors. E is the set of e directed arcs or edges between the tasks that maintain a partial order among them. The partial order introduces precedence constraints, i.e. if edge $e_{i,j} \in E$, then task v_j cannot start its execution before v_i completes. Matrix D of size $v \times v$ denotes the communication data size, where $d_{i,j}$ is the amount of data to be transferred from v_i to v_j. A task graph is a weighted graph. The weight w_i of a node v_i usually represents its computation cost. The weight of an edge stands for the communication requirement between the connected tasks (the amount of data that must be communicated between them). We introduce a new approach to assign node weight in Section 3.3.

In a given task graph, a root node is called an *entry task* and a leaf node is called an *exit task*. We assume that the task graph is a single-entry and single-exit one. If there is more than one exit or entry task, we can always connect them to a zero-cost pseudo exit or entry task with zero-cost edges. This will not affect the schedule.

3.2.2 Resource graph

A resource graph is an undirected weighted graph (both nodes and edges are weighted). A node of a resource graph represents a processor and an edge denotes the link between a pair of connected processors. The resource graph is a complete graph with p fully

connected nodes. The weight of a node represents the processor computation capacity (the amount of computation that can be performed in a unit time). Similarly, the weight of an edge stands for its communication capacity (the amount of data that can go through the link in a unit time). We further assume that all inter-processor communications are performed without contention. This assumption holds since our computing environment consists of processors connected with wide area network links as pointed out in [22].

3.2.3 Performance criteria

Before presenting the performance criteria, it is necessary to define a few attributes used in the algorithm. The computation cost of task v_i on processor p_j is $w_{i,j}$. If v_i cannot be processed on p_j, then $w_{i,j} = \infty$. The data transfer rates between processors are kept in a matrix dtr of size $p \times p$. The startup cost of communication for each processor is stored in a vector sc of size p. The communication cost $c_{i|m,j|n}$ from task v_i to v_j when task v_i is scheduled on processor p_m and task v_j is scheduled on processor p_n is given by

$$c_{i|m,j|n} = sc_m + \frac{d_{i,j}}{dtr_{m,n}} \tag{3.1}$$

We assume that intra-processor communication cost is negligible, i.e. $c_{i|m,j|m} = 0$. The task graph's edge weight is defined as the average communication cost:

$$c_{i,j} = \bar{sc} + \frac{d_{i,j}}{\bar{dtr}} \tag{3.2}$$

$EST(v_i, p_j)$ and $EFT(v_i, p_j)$ are the earliest execution start time and the earliest execution finish time of task v_i on processor p_j respectively. The entry task can start execution at time 0. Other tasks' EST can be computed by

$$EST(v_i, p_j) = \max\{avail(v_i, p_j), \max_{v_k \in pred(v_i)}(FT(v_k, p_{s_k}) + c_{k|s_k,i|j})\} \tag{3.3}$$

where $avail(v_i, p_j)$ is the earliest time at which processor p_j is ready for task v_i's execution; $pred(v_i)$ is the set of immediate predecessor tasks of task v_i. The inner max block in Eq. 3.3 is the time that all the data needed to execute task v_i on processor p_j is available, i.e. the *ready time*. This is obtained by considering all immediate predecessors of task v_i, the time they finish (FT) and the time needed to transfer data from the machine where they actually run on to the machine in consideration p_j. The EFT is defined by

$$EFT(v_i, p_j) = w_{i,j} + EST(v_i, p_j) \tag{3.4}$$

The schedule length L of the DAG is the actual finish time of the exit task v_{exit}.

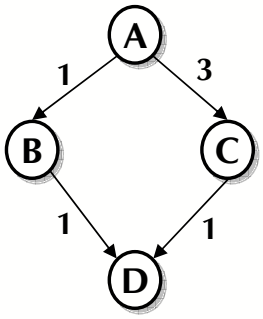
$$L = FT(n_{exit}) \tag{3.5}$$

Although several performance criteria such as the tardiness or the total flow time are suggested in the literature [21], our goal of scheduling in this research is to minimize the scheduling length L (*makespan*).

3.3 The SDC algorithm

3.3.1 Setting task node weight

There are various ways to set the weights of task nodes in a heterogeneous setting [109]. For instance, one can take the average value, the best value, etc. In our algorithm, we consider the effect of scarcity of resources as well as the average computation cost. We set relatively higher weight to the node with less capable resources. The rationale behind this is that tasks with scarce capable resources should be given higher priority in order to avoid situations that give rise to undesirable effects. This can be best illustrated with an example. In Fig. 3.1(a-c), an example task graph and its resource information are given. Task B and C have the same average computation cost. In addition, B can only be processed on processor 1. Fig. 3.2(a) shows the schedule when C is scheduled before B. In this case, task C has a smaller earliest finish time when it is assigned on processor 1. Task B has no choice but to be scheduled on processor 1. Assigning task C on processor 1 will delay the starting time of task B and therefore postpone the whole schedule. To avoid this problem, we can intentionally assign larger weights to tasks with few capable processors. As illustrated by Fig. 3.2(b), task B is considered before task C. B is still assigned on processor 1. This time C is scheduled on processor 2. The schedule length is reduced from 7 to 6 due to the change of scheduling order between tasks B and C.



(a)

Task	P1	P2
A	1	1
B	2	∞
C	3	1
D	1	1

(b)

	P1	P2
P1	∞	1
P2	1	∞

(c)

Figure 3.1: (a)an example DAG (b)the computation cost for each node on three machines (c) the communication cost table

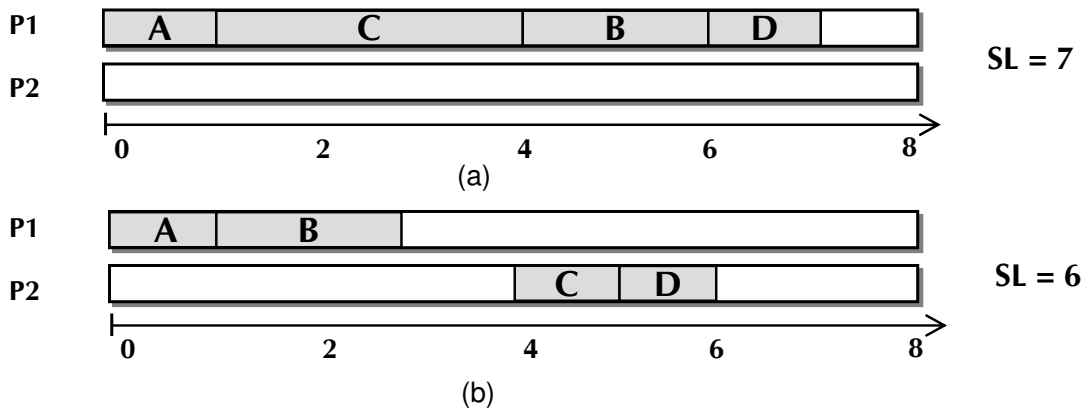


Figure 3.2: (a)schedule for the DAG in Fig.3.1 with priority list A,C,B,D (b)schedule for the DAG in Fig.3.1 with priority list A,B,C,D

The set of capable processors for node v_i is denoted as

$$CP(v_i) = \{p_k | w_{i,k} \neq \infty\} \tag{3.6}$$

We define the *Percentage of Capable Processors* (PCP) of node v_i as

$$PCP(v_i) = \frac{||CP(v_i)||}{p} \tag{3.7}$$

where p is number of all processors. Thus the *Percentage of Incapable Processors* (PIP) is

$$PIP(v_i) = 1 - PCP(v_i) \tag{3.8}$$

The weight of node v_i is specified as

$$w_i = \frac{\sum_{p_j \in CP(v_i)} w_{i,j}/||CP(v_i)||}{PCP(v_i)} \tag{3.9}$$

By applying this specification we give relatively higher weights and thus higher priorities to those task nodes with fewer capable resources. Experimental results in Section 3.4 show that this method gives better schedules than the one using average computation costs.

3.3.2 Prioritizing the tasks

This step is essential for list scheduling algorithms. A task processing list is generated by sorting the task by decreasing order of some predefined rank function. In this research, we use *b-level* [65] as the rank function. The *b-level* of node v_i is the length of the longest path from v_i to the exit node. It can be obtained by recursively traversing the task graph from the exit node with time complexity $O(e + v)$.

$$BLEV(v_i) = w_i + \max_{v_j \in succ(v_i)} \{\overline{c_{i,j}} + BLEV(v_j)\} \tag{3.10}$$

where $\overline{c_{i,j}}$ is the average communication cost of $e_{i,j}$, w_i is the weight of node v_i, and $succ(v_i)$ is the set of immediate successors of v_i. Ties are broken randomly in order not to introduce high computing cost. The sorted list preserves the precedence constraints among tasks.

3.3.3 Selecting processors

Various criteria have been proposed to select suitable processor for a task. When scheduling in a homogeneous environment, EST is a popular choice [1, 51, 106]. While in heterogeneous settings, using EFT as selection criteria gives better schedules [98]. Sih and Lee [94] suggested selecting (node, processor) pairs that maximize the so-called *Dynamic Level* at each step. They extended the definition of *Dynamic Level* by including the effects of descendant and resource scarcity when scheduling in heterogeneous systems. Furthermore, an insertion based policy is better than non-insertion based one

as observed in [64]. An insertion based policy considers scheduling idle time slot between two already scheduled nodes as long as the slot is long enough and inserting the task to the slot does not break any precedence constraint.

The EFT method apparently fails to consider how well the descendants of current scheduling node v_i matches the selected p_j that minimizes the $EFT(n_i, p_j)$. This is of particular importance in our computing environment where processors have different capabilities as identified in [94]. We propose a new target function called *Adjusted Earliest Finish Time* (AEFT). The SDC algorithm assigns task to the processor that minimizes the AEFT with an insertion-based policy. The $AEFT$ is defined as

$$AEFT(v_i, p_j) = EFT(v_i, p_j) + \frac{1}{||succ(v_i)||} \sum_{v_t \in succ(v_i)} \sqrt[s_t]{\prod_{w_{t,k} \neq \infty} c_{i|j,t|k}} \tag{3.11}$$

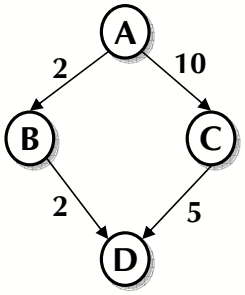
where $s_t = ||CP(v_t)||$ is the number of capable processors for task v_t. For each child v_t of v_i, we calculate the geometric average of its communication cost with v_i (assuming it is scheduled on p_j) when v_t is scheduled on each capable processor $p_k \in CP(v_t)$. The second term in Eq. 3.11 considers how the current node's allocation will affect the communication with its descendant, on average. Without the second term, undesirable results can be produced. For example, in the case where $EFT(v_i, p_j)$ is minimized, due to the scarcity of capable processors to execute v_i, it has to be placed on some processor, say p_k, where communication cost between p_j and p_k can be very expensive. This will undermine the overall quality of the resulting schedule.

(1)Set the weights of task nodes with Eq.(3.9)
(2)Set the weights of edges with Eq.(3.2)
(3)Compute the *b-level*s for all tasks by traversing graph
upward from the exit node.
(4)Sort the task into a list by non-increasing order of b-level
(5)***while*** the scheduling list is not empty ***do***
(6) Remove the first task v_i from the list for scheduling
(7) ***for*** each processor capable p_j of v_i ***do***
(8) Compute $AEFT(n_i, p_j)$ value with Eq.(3.11)
using insertion-based policy
(9) ***endfor***
(10) Assign task v_i to the processor that minimize $AEFT$ of v_i
(11)***endwhile***

Figure 3.3: The SDC algorithm

3.3.4 Procedure of the algorithm

The pseudo-code of the SDC algorithm is shown in Fig. 3.3. As with other list scheduling algorithms, the SDC algorithm has two major stages; a *task prioritizing stage* and a *processor selection stage*. The first stage computes the priorities of all the tasks while the second one selects the tasks in the order of their priorities and assigns each selected task on its most desirable processor, which minimizes the task's adjusted finish time. As an illustration, Fig. 3.4(a) presents a sample DAG. The number next to each edge of the graph corresponds to the amount of data that needs to be transferred from a task to its immediate successor. The cost to execute each of the four tasks in the graph on each of three different machines is given in Fig. 3.4(b). Fig. 3.4(c) shows the cost to transfer a data unit for any pair of machines. For the simplicity of illustration, we use unit data transfer rate.



Task	P1	P2	P3
A	1	∞	2
B	2	∞	1
C	∞	2	4
D	5	3	∞

	P1	P2	P3
P1	∞	1	1
P2	1	∞	1
P3	1	1	∞

(a) (b) (c)

Figure 3.4: (a)an example DAG (b)the computation cost for each node on three machines (c) the communication cost table

Fig. 3.5 shows the schedules obtained by HEFT, DLS and our SDC algorithm. The schedule length of SDC is shorter than those of the other two algorithms. The scheduling list of HEFT and SDC happen to be the same, which is $\{A, C, B, D\}$. DLS algorithm does not maintain a static scheduling list. It selects a pair of (node, processor) that maximize the *Dynamic Level* at each step.

3.3.5 Time-complexity analysis

We will refer to Fig. 3.3 when analyzing the time complexity of the algorithm. Line (1) and (2) take $O(vp)$ time. Line (3) can be done in $O(e + v)$ [65]. Sorting tasks in Line (4) takes at most $O(v \log v)$. Line 5 - 11 will cost $O(ep^2)$ time. Thus the total time complexity is $O(ep^2)$. For a dense graph, where e is proportional to $O(v^2)$, the time complexity becomes $O(v^2p^2)$.

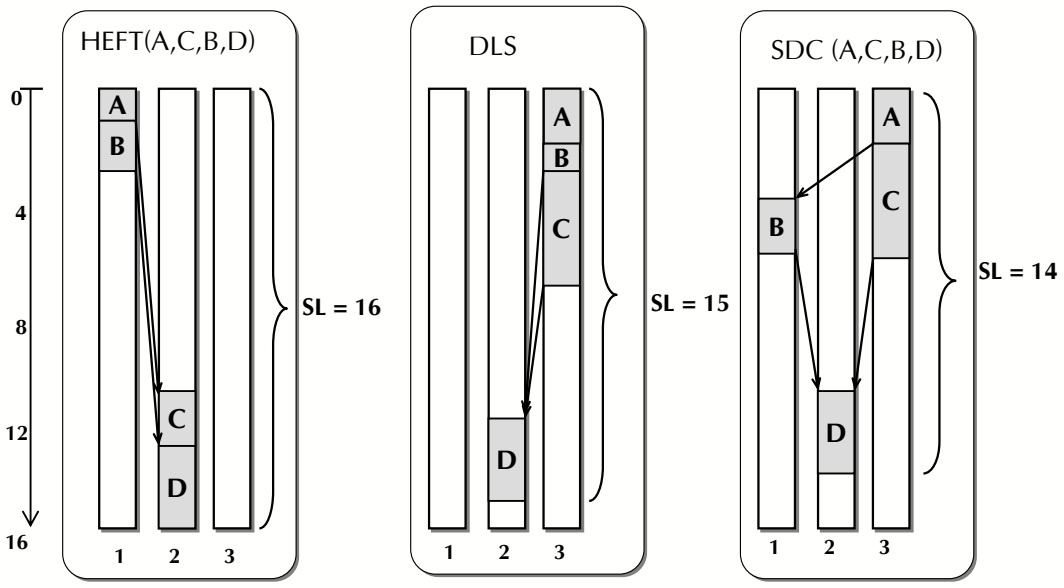


Figure 3.5: (a)HEFT algorithm (b)DLS algorithm (c)SDC algorithm

3.4 Experimental results and discussion

We have evaluated our algorithm with a wide range of graphs. In this section, we present the comparative results of the SDC algorithm and some related work given in Chapter 2, namely HEFT and DLS. Randomly generated DAGs and a genomic sequence annotation workflow are considered for assessing the algorithms.

3.4.1 Comparison metrics

The comparisons of the three algorithms are made using the following two measures:

- *Normalized schedule length*(NSL). The principal performance metric of an algorithm is the length of its output schedule. The NSL of an algorithm is defined

as:

$$NSL = \frac{L}{\sum_{v_i \in cp_{min}} \min_{p_j \in P} \{w_{i,j}\}} \tag{3.12}$$

where L is the schedule length. cp_{min} is the critical path of the DAG when the task node weights are evaluated as the minimum computation cost among all capable processors. The denominator represents a lower bound on the schedule length. Such a lower bound may not always be possible to reach and $NSL \geq 1$ for any algorithm. We use averaged NSL over set of DAGs as a comparison metric.

- *Average Percentage Degradation* (APD). The APD of an algorithm is the average (over all DAGs) of the percentage of degradation of the schedule lengths L produced by the algorithm from the best schedules. Let G denote a set of DAGs, where $G = \{g_1, g_2, ...\}$. $ALG = \{alg_1, alg_2, ...\}$ is the set of algorithms we are comparing. $sl(alg_i, g_j)$ represents the schedule length of g_j using algorithm alg_i. The APD of algorithm alg_i over graph set G is defined as:

$$APD(alg_i, G) = \frac{\sum_{g_j \in G} \left(sl(alg_i, g_j) - sl\left(\underset{alg \in ALG}{\operatorname{argmin}} \left(sl(alg, g_j) \right), g_j \right) \right)}{||G||} \tag{3.13}$$

- *Efficiency.* The *speedup* of a task graph is defined as the time required for sequential execution of the graph in a single processor, divided by the time it takes to complete it with N processors. We assume that there is at least one processor that can execute all the tasks when comparing efficiencies of various algorithms.

The sequential execution time is obtained by assigning all tasks to a single processor that minimizes the cumulative computation costs. *Efficiency* is the ratio of speedup to the number of processors used. We use efficiency as the metric to test the scalability of our algorithm. The results are presented in Section 3.4.2.

3.4.2 Randomly generated application graphs

Random task graph generation

In the first part of evaluation, task graphs are generated randomly with the following input parameters:

- Task size in the graph (v). The value of v is assigned from the set $\{20, 40, 60, 80, 100\}$.
- Shape parameter of the graph (α). The height of a DAG is $h = \frac{\sqrt{v}}{\alpha}$. α gets its value from set $\{0.5, 1.0, 2.0\}$.
- Average computation cost($\overline{comp}$). The average computation cost of a task node is the average time required to complete the task on all of its capable processors. The average computation cost of task node v_i ($\overline{comp}_i$) is generated randomly from normal distribution $N(\overline{comp}, 0.5\overline{comp})$. Then the computation cost of v_i on processor p_j ($w_{i,j}$) is from normal distribution $N(\overline{comp}_i, 0.5\overline{comp}_i)$. The values of $\overline{comp}$ is from set $\{10, 20, 30, 40, 50\}$.
- Communication-to-Computation Ratio(CCR). The graph's CCR is the ratio of average communication cost to the average computation cost. $CCR = \{0.01, 0.1, 1.0, 10, 100\}$.

- Average communication cost($\overline{comm}$).

$$\overline{comm} = CCR * \overline{comp} \tag{3.14}$$

- Percentage of Incapable Processors(PIP). This is defined in Eq. 3.8. There are two schemes used when setting PIP. In the first set of experiments, we investigated the effect of a new weight assignment function on schedule length. The PIP of each task node is randomly generated from uniform distribution (0,0.9). We want to evaluate how the SDC algorithm performs with respect to PIP in the second set of experiments. The PIP of task node v_i,$PIP(v_i)$ is from normal distribution $N(\overline{PIP}, 0.5\overline{PIP})$, where $\overline{PIP} = 0, 0.1, 0.2, ..., 0.9$.

Three sets of experiments are conducted in this part of the evaluation. Experiment set I are designed to examine the effectiveness of the weight assignment function described in Section 3.3.1. Experiment set II assesses the validity of the processor selection criteria outlined in Section 3.3.2. When generating the graphs, each parameter set is repeated 25 times for the first set of experiments and 10 times for the second. This gives 9,375 graphs for Experiment set I and 37,500 graphs for Experiment set II. Experiment set III evaluates the efficiency of the algorithm. The processor number varies from 4 to 64. Other parameters are the same as those of experiment set I.

Generation of resource graph

The resource graph as described in 3.2.2 is a complete graph. The parameters we need to set are:

- Number of processors (p).
- Average data transfer rate ($\overline{dtr}$). This the average data transfer rate over all combinations of processors. We fix this value as 1. The data transfer rate between p_m and p_n ($dtr_{m,n}$) is from normal distribution $N(1, 0.5)$. We only use numbers that are positive.
- Average data transfer size ($\overline{d}$). Since the average data transfer rate is 1, the average data transfer size is the same as the average communication cost. The data size to be transferred from task v_i to v_j is $d_{i,j} \sim N(\overline{dtr}, 0.5\overline{dtr})$.
- Startup cost. In this study, we omit the startup cost.

Performance comparison

The algorithm presented in Sec. 3.3 has two distinctive features; a new weight assignment method and a modified processor selection criteria. The effects of both features are presented next.

Effect of weight assignment function Experiment set I investigates how the weight assignment method will impact the average NSL and APD. The results are shown in Figs. 3.6 and 3.7 respectively. Two algorithms, namely HEFT and DLS, and their

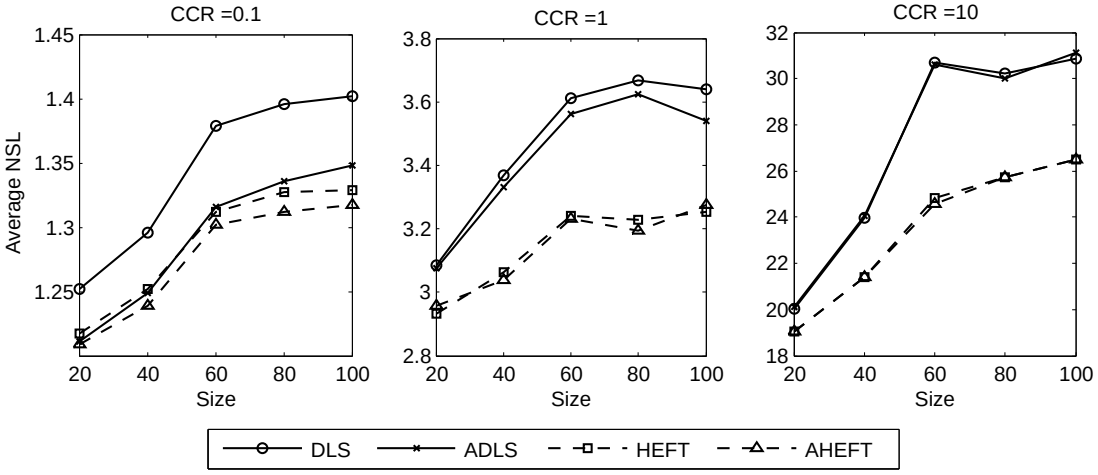


Figure 3.6: The effect of the weight assignment method on average NSL

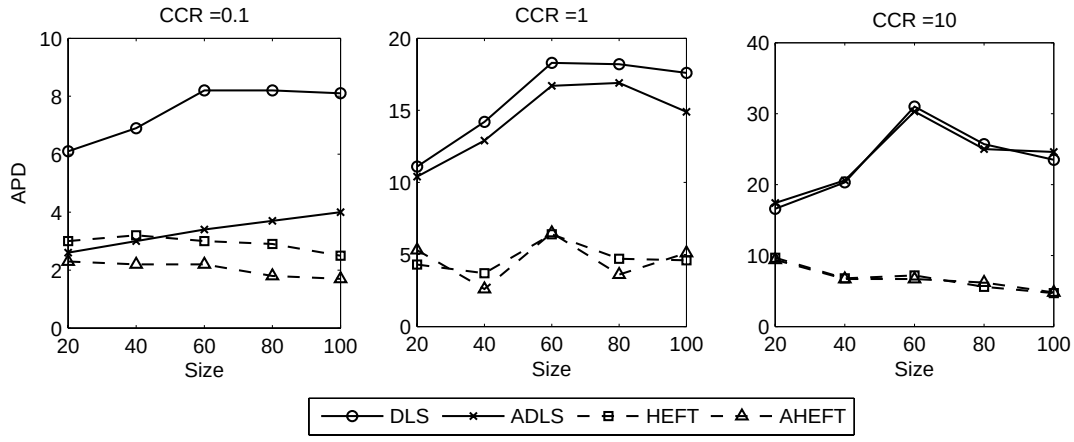


Figure 3.7: The effect of the weight assignment method on average percentage degradation

modified counterparts AHEFT and ADLS are compared. The AHEFT (respectively ADLS) is adapted from HEFT (DLS) where our weight assignment method described in Section 3.3.1 is adopted. We show the cases where CCR =0.1, 1 and 10. From Fig. 3.6 we first observe that for all cases the average NSLs show an increasing trend with respect to the increase of task graph size. This is due to the fact that the proportion of task nodes, other than those on the critical path, increases with the task graph size making it more difficult to achieve the lower bound.

We also notice that the adjusted algorithm performs better than its corresponding original version and the degree of improvement varies with respect to CCR. When CCR=0.1, the improvement of modified DLS over DLS is 5.0% when task size is 100. The average NSL is reduced by 2.2% in the case of AHEFT (the adjusted HEFT algorithm) versus HEFT. When CCR increases to 10, there is no noticeable effect. Remember that we use *b-level* (defined in Eq. 3.10) as the priority of a task node. When CCR is large, average communication cost dominates $BLEV$ in Eq. 3.10. Adjustment of the task node weight does not really impact the priority, thus does not affect the scheduling list order. On the other hand, when CCR is small, assigning a higher weight to those task nodes with large PIP can give higher priority to the tasks and therefore change the scheduling list order. As a result the schedule length is improved.

Fig. 3.7 depicts the degradation from the best solutions of the algorithms. When CCR=0.1, the APD of AHEFT is less than 2.1% for all task graph sizes. On average, the APD of DLS is improved by 4.8% when CCR=0.1. From the first graph of Fig.3.7,

we notice that the decrease of APD is more perceivable for DLS than that for HEFT. In the HEFT algorithm, the priority of each task node is set at the beginning and the scheduling list remains unchanged during the whole procedure. However, the DLS algorithm reevaluates the *dynamic level* at each scheduling step and selects the (ready node, processor) pair that maximizes it. The task weight constantly affects the *dynamic level* values during the scheduling process.

Effect of processor selection criteria Experiment set II validates the processor selection policy. We investigate how the algorithms will be impacted under graphs with various characteristics.

Fig. 3.8 gives the average NSL values of the algorithms at different CCR, task size and PIP. The DLS, HEFT and NSDC are the algorithms without a weight assignment adjustment. When comparing three figures on each row, we notice that the average NSLs tend to increase with the increasing of task graph size. This is consistent with previous observations in the first set of experiments.

When CCR is small, the NSDC algorithm performs almost the same as HEFT. This is because the second term of the definition for AEFT (Eq. 3.11) is relatively small compared to EFT. However, due to the significant impact of the weight assignment function in the cases of small CCR, SDC generates better schedules overall. This is more obvious when $PIP > 0.4$. The average NSL of SDC algorithm is better than the DLS algorithm by 4.7%, the HEFT algorithm by 1.6% when CCR = 0.01.

When CCR is large, the communication cost becomes dominant. Scheduling the

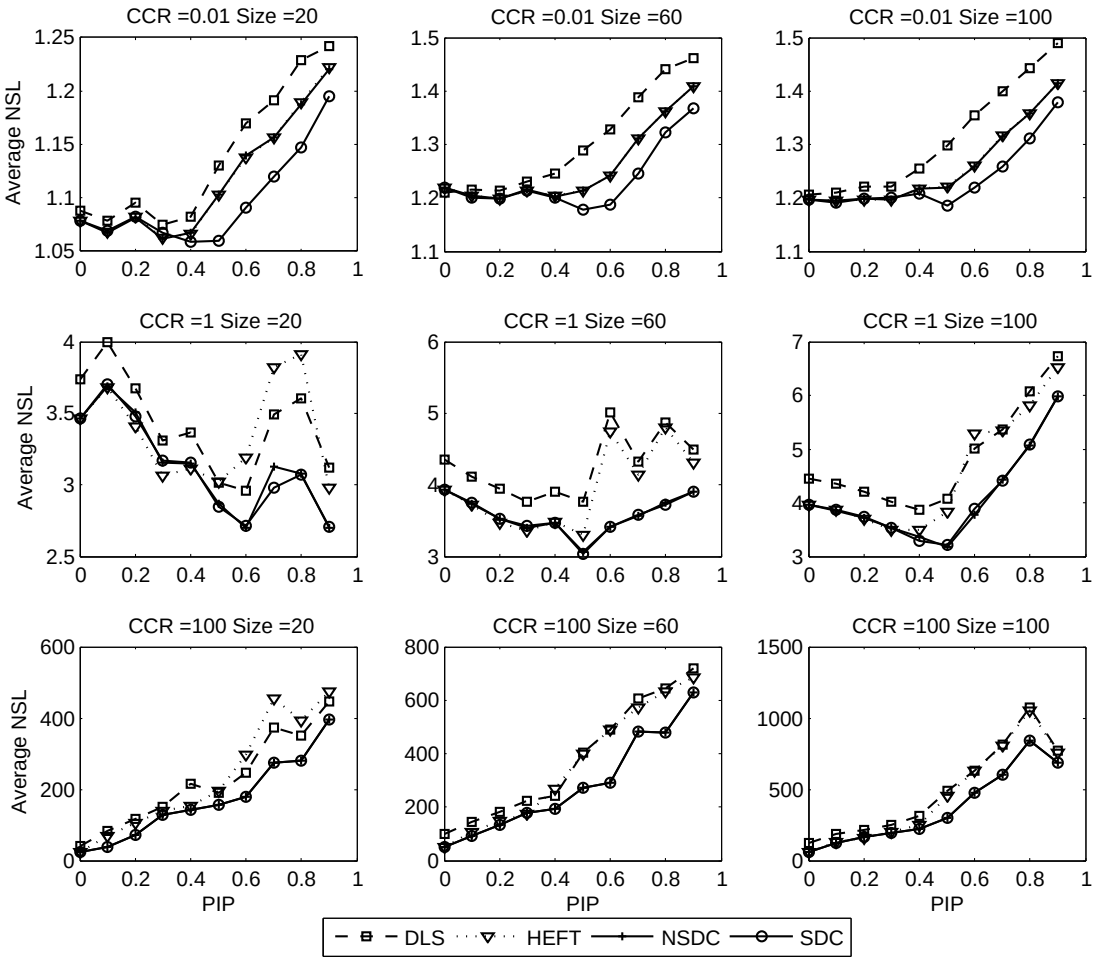


Figure 3.8: Average NSL of the algorithms

task without considering communication cost will suffer a huge performance penalty if the children of the node can only be processed on a subset of available resources. The average NSL of SDC is smaller than the DLS algorithm by 16.4%, the HEFT by 28.3% when CCR = 100. It is also noticed that when CCR is large, the average NSL is large in that the lower bound of schedule length does not include any communication cost. Our algorithm works better overall, especially when the heterogeneity of processor capabilities is considerable.

The *Average Percentage Degradation* (APD) of the algorithms at different parameters is given in Fig. 3.9. It can be seen that in almost all cases the APD of our algorithm remains the lowest. The APDs of the other two algorithms fluctuate with respect to both PIP and CCR. This indicates that our algorithm is less sensitive to PIP and CCR compared to the other two.

Efficiency comparison We compared the efficiency of three algorithms, namely, SDC, HEFT and DLS. The number of processors used varies from 4 to 64, incrementing by a power of 2. The rest of parameters are the same as those used previously. Fig. 3.10 shows the comparison with graph size 100 and CCR 0.1. SDC has consistently better efficiency than the other two algorithms. HEFT and DLS have comparable efficiency when the processor number is small. As the number of processors increases, HEFT surpasses DLS with respect to efficiency.

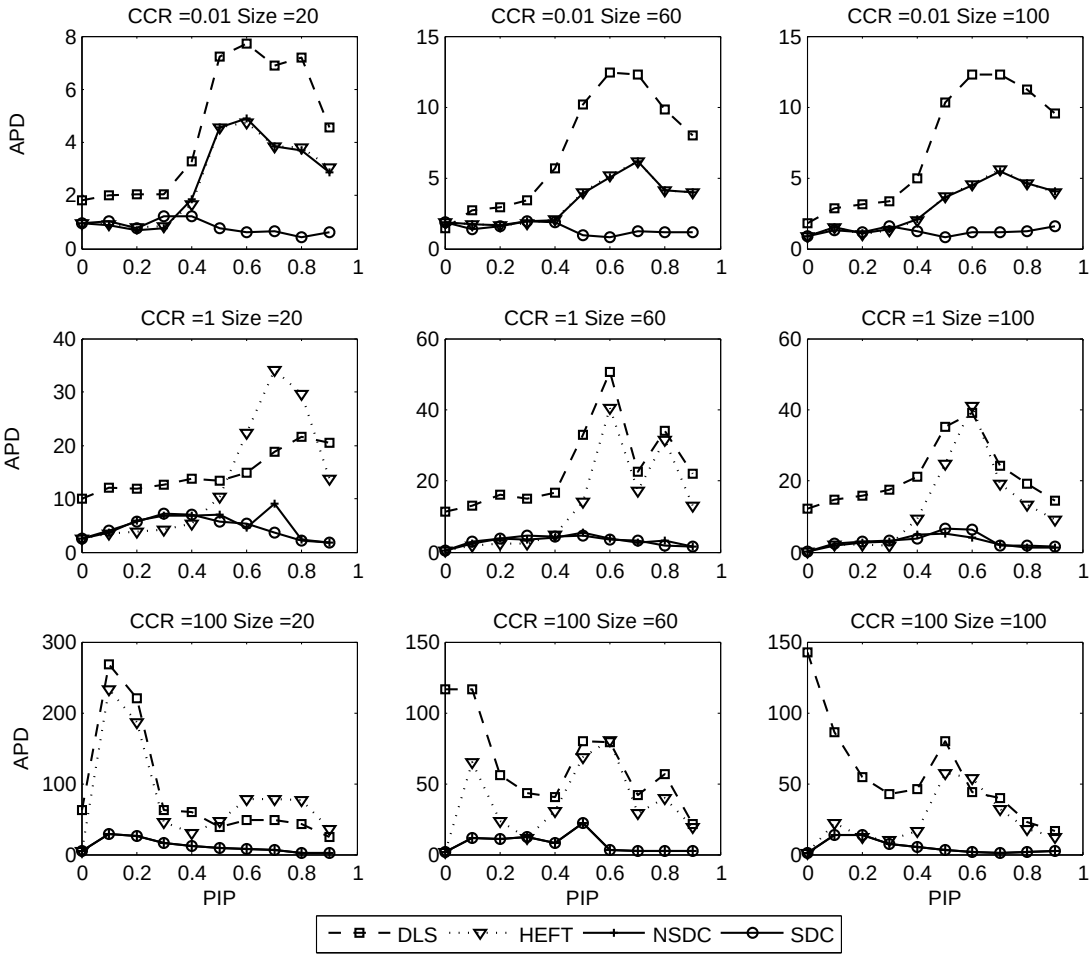


Figure 3.9: Average percentage degradation of the algorithms

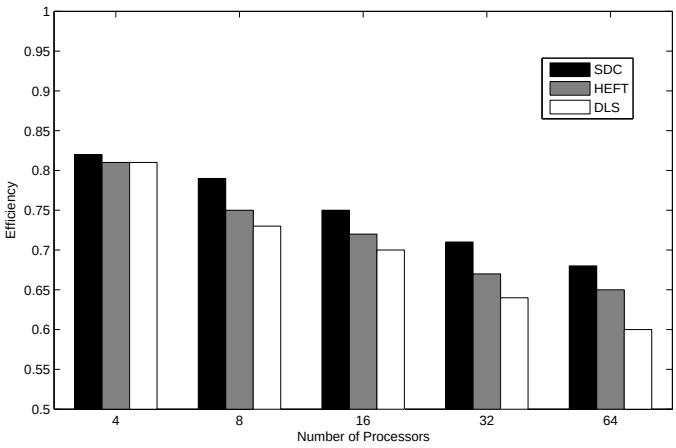


Figure 3.10: Efficiency comparison with respect to the number of processors

3.4.3 Performance analysis on application graph of a genomic sequence annotation workflow

We further tested our algorithm with a genomic sequence annotation workflow [91] . Fig. 3.11 shows the task graph. In the figure, the DAG branches after the input sequence *File* node into a sub-DAG of analysis that work on the original input and a sub-DAG that analyzes the input sequence that is masked for repeats with *RepeatMasker*. The unmasked sequence is analyzed further using three software packages, namely *tRNAscanSE*, *Genscan* and *HmmGene*. The masked sequence is searched against two databases using *Blastall*. The results from the latter search are further processed by an application (*bt2fasta*). This generates a new database of formatted gene sequences. The unmasked input sequence is then used as input to *Sim4*, which in turn aligns the input sequence to

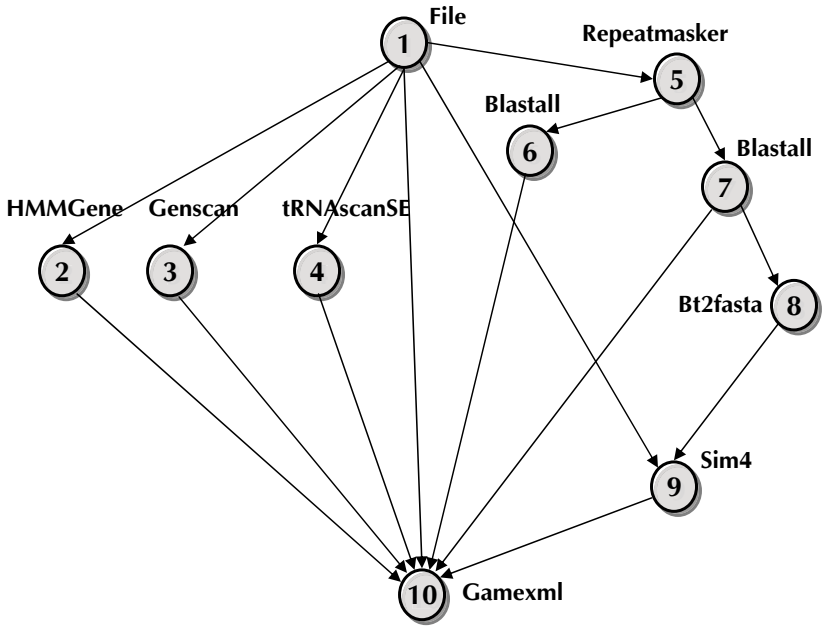


Figure 3.11: A genomic sequence annotation workflow

the entries in the newly created database. Results for all analyses are then integrated into an XML file for further interpretation using some annotation tool. In this workflow application, several domain specific softwares are involved. Because of these softwares' special requirements, some of them can only be installed on designated machines while others are available on all processors. This is a good example where processors offer different capabilities in terms of software availability.

There are 10 tasks and 16 edges in the graph. We set the relative computation units according to the tasks' demands. The transferred data size is also specified approximating the corresponding file size. Processor number varies between 2 and 10. The PIP of each processor is set randomly. Fig. 3.12(a) shows the performance of the algorithms

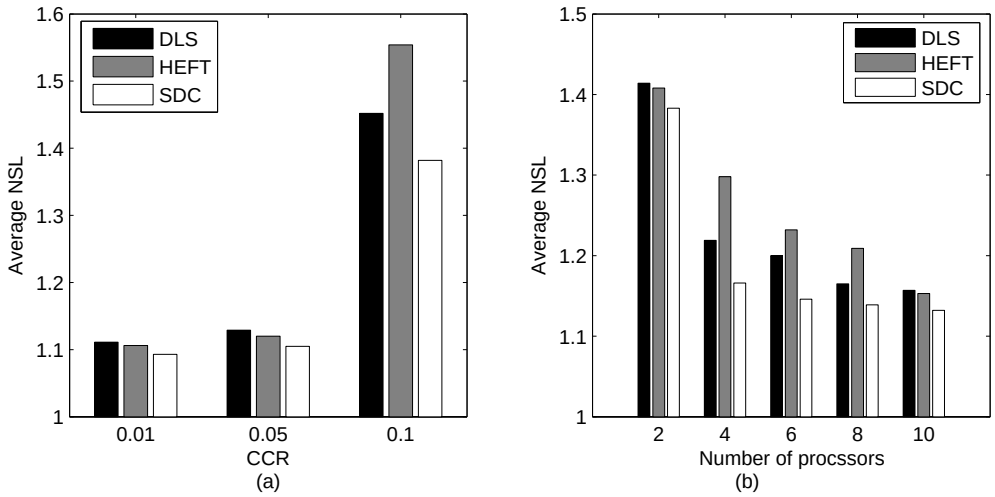


Figure 3.12: Comparison of three algorithms on a genomic sequencing annotation workflow

with respect to three different CCR values. The highest CCR is set to 0.1 because the workflow is computation-intensive in reality. On the average, SDC performs best among the three algorithms. The performance gain is more notable for larger CCRs. Compared to DLS, HEFT produces a better schedule when CCR is small. When CCR increases to 0.1, the trend is reversed. In [98], where all the processors can handle every task, the conclusion is different. The authors observed that HEFT always obtains a smaller average NSL when testing with a modified molecular dynamic task graph. The comparison of three algorithms with regard to different processor numbers is given in Fig 3.12(b). It is noticed from the figure that since there are at most four tasks in any level in the task graph, increasing the processor number does not significantly reduce SLR if $p > 4$. The SDC algorithm outperforms the other two algorithms in all cases.

3.5 Conclusions

In this chapter, we presented a new algorithm for scheduling DAG based workflow applications in heterogeneous systems where processors have different capabilities. The algorithm has two distinctive features. First, we consider the effect of tasks' scarcity of capable processors when assigning the task node weights. For two task nodes with the same average computation cost, our weight assignment policy tends to give higher weight to the task with large PIP. Secondly, during the processor selection phase, we adjust the effective EFT strategy by incorporating the average communication cost between the current scheduling node and its children. We evaluate the algorithm using a large set of randomly generated task graphs with different characteristics and a real world bioinformatics workflow application. Results show that each feature of the SDC algorithm improves the schedule length. It is noted that the new weight assignment policy perceivably impacts the schedule when CCR is small while the processor selection strategy affects the schedule length more substantially at a larger CCR. By combining the two strategies, the SDC algorithm outperforms the other two algorithms overall. Efficiency comparisons among three algorithms reveals that SDC scales well for various processor numbers.

Chapter 4

Robust Task Scheduling in Non-deterministic HDCS

4.1 Introduction

Although differing in the ways of modeling target computing systems (e.g., heterogeneous vs. homogeneous processors, with vs. without communication cost ,etc.), most traditional scheduling methods are based on a *deterministic* model. In this model, all information about the tasks (durations) and relationships among them (dependencies in the DAG) are supposed to be known by the scheduling algorithm *a priori*. It is assumed that the task execution time can be estimated and does not change during the course of execution. However, this assumption does not usually hold in a real computing environment, where the actual execution time of a task is different from the expected

one. The problem can be dealt with in several ways. For example, dynamic scheduling algorithm assigns each ready task according to the current status of the resource environment, aiming to avoid the inaccuracy of execution time estimation. Another possible approach is to judiciously overestimate the execution time of each task according to its variability, hoping that the real execution time will not exceed the estimated one. Thus, the schedule will perform as well as expected. However, this approach could result in low resource utilization. In this chapter, we take on the challenge by using a static algorithm to find schedules less vulnerable to the non-deterministic nature of the task execution time, i.e., more robust. As with other deterministic scheduling algorithms, our scheduler is fed with the expected task execution times. We then define a metric called *slack* for a schedule based on the slack of the individual task. The slack of a task represents a time window within which it can be delayed without extending the makespan and it is intuitively related to the robustness of the schedule. Larger slack tends to absorb the task execution time variance with little delay. Next, we develop a genetic algorithm based heuristic to generate schedules that are more robust compared with schedules obtained by another popular heuristic called HEFT [98]. Genetic based task scheduling algorithms [25, 50, 100, 105] normally use the *makespan* as their objective function. However, in order to take into account both the robustness and makespan, it is necessary to include the slack in the objective function. Unfortunately, slack and makespan are two conflicting metrics as shown in section 4.4.1. Optimizing only the makespan will result in schedules with small slack thus less robust to task

execution time variability. Conversely, optimizing slack alone tends to give a robust schedule but with large makespan. To handle this multi-objective optimization problem (MOOP) [32], we employed the *ϵ-constraint method*. In this method, an upper bound of expected makespan is given by $\epsilon \cdot \text{Makespan}_{HEFT}$. The scheduling algorithm tries to find the schedules with maximal slack without exceeding the specified upper-bound. Although the robustness of a schedule is a desirable property and conceptually easy to perceive, it is difficult to measure quantitively. There are several attempts to define it according to different perspectives of the problem [5, 7, 37, 66]. We give two new measures of robustness based on tardiness and miss rate in this work. Results show that the proposed algorithm can effectively trade off makespan for robustness.

The rest of this chapter is organized as follows. In Section 4.2 the robust task scheduling problem is described. Section 4.3 presents a genetic algorithm based approach to solve the bi-objective optimization problem. We show some experimental results in Section 4.4. The chapter concludes in Section 4.5.

4.2 Robust task scheduling problem

In this section, we present a formulation of robust scheduling a task graph.

4.2.1 Basic Models

As in Chapter 3, a task graph is defined by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_1, v_2, ..., v_n\}$ is the set of n tasks. $\mathcal{E}$ is the set of directed arcs or edges between the tasks that maintain

a partial order among them. The partial order introduces precedence constraints, i.e., if edge $e_{i,j} \in \mathcal{E}$, then task v_j cannot start its execution before v_i completes. v_i is an *immediate predecessor* of v_j, and v_j is an *immediate successor* of v_i. A node with no predecessor is called an entry node, and a node with no successor is called an exit node. Matrix $\mathcal{D}$ of size $n \times n$ denotes the communication data size, where $d_{i,j}$ is the amount of data to be transferred from v_i to v_j. A heterogeneous multiprocessor computing system is composed of a set $\mathcal{P} = \{p_1, p_2, ..., p_m\}$ of m fully connected processors. We assume that all inter-processor communications are performed without contention and computation can be overlapped with computation. To each task v_i, there is an associated vector representing its minimal duration on each processor, i.e., the best case execution time ($BCET$). $\mathcal{B}$ is an $n \times m$ matrix where $b_{i,j}$ gives the best case execution time of task v_i on processor p_j. Furthermore, we assume that random variables $c_{i,j}$ are independent of each other. The data transfer rates between processors are represented by matrix $\mathcal{TR}$ of size $m \times m$. Intra-processor communication cost is assumed to be zero. In this work, we do not consider the variation in data transfer rates.

A *schedule* represents the assignment of tasks onto processors. It is denoted as a vector $s = \{s_1, s_2, ..., s_m\}$, where $s_i = \{(v_{j_1}, v_{j_2}), ..., (v_{j_{k_i-1}}, v_{j_{k_i}})\}$ denotes the task execution order on processor i. k_i is the number of task nodes assigned to processor i. Fig. 4.1 illustrates an example of a task graph, a multiprocessor system and a schedule. The schedule shown in Fig. 4.1(c) can be denoted as $\{\{(v_1, v_2), (v_2, v_4)\}, \{(v_3, v_5), (v_5, v_8)\}, \{(v_6, v_7)\}, \phi\}$.

Definition 4.2.1. *Given a task graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *and a schedule* $s = \{s_1, s_2, ..., s_m\}$,

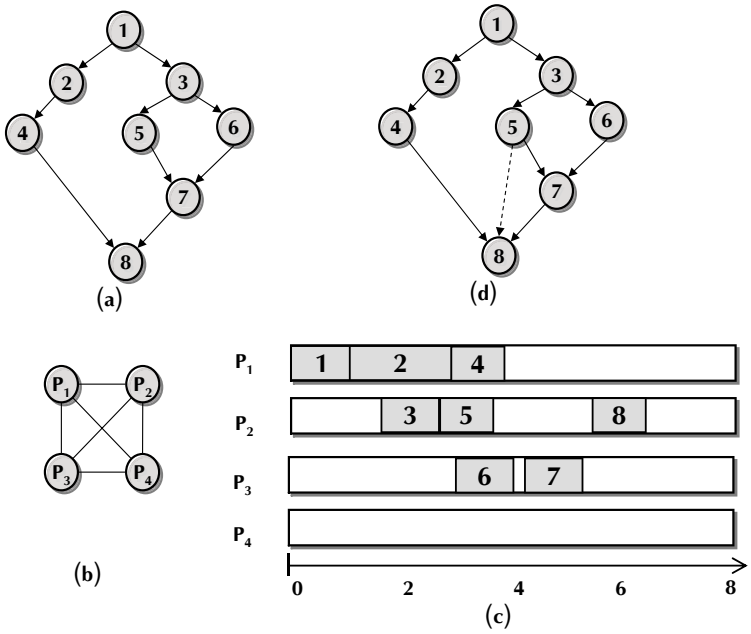


Figure 4.1: (a) An example task graph (b) A multiprocessor system (c) A schedule (d) A disjunctive graph of (a) with schedule (c)

we denote by $\mathcal{G}_s$ *the disjunctive graph of* $\mathcal{G}$ *under schedule* s *as:* $\mathcal{G}_s = (\mathcal{V}_s, \mathcal{E}_s)$*, where* $\mathcal{V}_s = \mathcal{V}$*,* $\mathcal{E}_s = \mathcal{E} \bigcup \mathcal{E}'$*.* $\mathcal{E}'$ *is the set of disjunctive edges.* $\mathcal{E}' = \{e_{i,j} | e_{i,j} \notin \mathcal{E}, \exists k \in \{1, ..., m\}, s.t. (v_i, v_j) \in s_k\}$*. The data size matrix associated with* G_s*,* $\mathcal{D}_s$ *is:*

$$d_{s,ij} = \begin{cases} 0 & \exists k \in \{1, ..., m\}, s.t. (v_i, v_j) \in s_k \\ d_{ij} & otherwise \end{cases} \tag{4.1}$$

Fig. 4.1(d) represents the disjunctive graph of (a) with the schedule shown in (c). Here, $\mathcal{E}'$ is illustrated with dashed line.

In traditional list scheduling algorithms such as those proposed in [35, 94, 98, 106],

a task is assigned to a "best" processor one at a time according to a pre-computed order. After the last task is scheduled, its finish time becomes the makespan of the whole schedule. There is only *one* makespan value for each schedule. This is not the case in robust task scheduling where task execution time variation is considered (in our experiments, the actual execution times will be modeled by random variables). We call it a *realization* of a schedule when the task graph is executed in the real resource environment according to the schedule. Clearly, each realization of a schedule can result in different makespans. The following claim states how we can obtain the actual makespan of a schedule.

Claim 4.2.2. *Given task graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *and a schedule* $s = \{s_1, s_2, ..., s_m\}$*, if each task starts to execute as soon as it becomes ready, then the makespan corresponding to the schedule* s *is the length of the critical path of disjunctive graph* $\mathcal{G}_s$*.*

4.2.2 Slack

Using the concept of slack to manage uncertainty in scheduling originates from the field of operations research. In [66], the authors used several surrogate measures based on average slack time to generate schedules that are robust to machine breakdown and processing time variations. Recently, Bölöni et al. [16] used slack to identify safe components in DAG scheduling. A safe component will not cause an increase in the total makespan.

We first introduce the following notations:

- $\mathcal{P}_{i\to j}$ the set of the paths that go from node i to node j in G',
- t_{s_i} the start time of task i,
- t_{f_i} the finish time of task i,
- T_i the duration of task i. $T_i = t_{f_i} - t_{s_i}$,
- Node m is the exit node. Node 1 is the entry node,
- $M = t_{f_m}$ the makespan,
- $C_{i,j}$ is the communication cost between task node i and j,
- $\mathrm{Bl}(i) = \max_{\mathcal{P}_{i\to m}} \left(\sum_{j=i}^{m-1} (T_j + C_{j,j+1}) \right) + T_m$, the bottom level of node i,
- $\mathrm{Tl}(i) = \max_{\mathcal{P}_{1\to i}} \left(\sum_{j=1}^{i-1} (T_j + C_{j,j+1}) \right)$, the top level of node i,
- $S_{i\to j} = t_{s_j} - t_{f_i} - C_{i,j}$ the spare time between i and j (when the edge (i,j) does not exist in G' we have $S_{i\to j} = 0$).

We assume that the schedules are *eager*, which means that once allocated a processor, each task starts execution as soon as possible. We propose to use the following as the definition of slack for a task node:

Definition 4.2.3. *Consider a task graph G, and a schedule s for the task graph. The makespan of G under schedule s is M. For a task node n_i, let $Tl(i)$ denote its top level, which is the length of a longest path from an entry node to n_i (excluding n_i). The length of a path is the sum of all the expected costs of nodes and edges along the path, once the*

schedule is computed. Let $Bl(i)$ stand for its bottom level. The bottom level of a node n_i is the length of a longest path from n_i to an exit node [65]. Then the slack of n_i is defined as:

$$\sigma_i = M - Bl(i) - Tl(i) \tag{4.2}$$

The average slack of task graph G is:

$$\sigma = \sum_{i=0}^{n-1} \sigma_i / n \tag{4.3}$$

where n is the size of task graph.

We first show that definition (4.2) is equivalent to the one proposed in [16]. In [16] the slack of task i was defined as:

$$\sigma'_i = \min_{\mathcal{P}_{i \to m}} \left(\sum_{j=i}^{m-1} S_{j \to j+1} \right)$$

Theorem 4.2.4. $\sigma_i = \sigma'_i$.

Before proving the above theorem, we need the following lemma:

Lemma 4.2.5. *For all tasks i we have $Tl(i) = t_{s_i}$.*

Proof of lemma 4.2.5 We shall prove this lemma by induction from the top of the graph. Since the schedule is *eager*, we have: $t_{s_1} = \mathrm{Tl}(1) = 0$. Suppose that it is true for all the parents i of a task j. Let i^* be a parent of j where the longest path from 1 to j

goes through. we have:

$$\mathrm{Tl}(j) = \mathrm{Tl}(i^*) + C_{i^*,j} = t_{s_{i^*}} + C_{i^*,j}$$

We have $t_{s_j} \geq \mathrm{Tl}(j)$ because we need to respect the dependencies between i^* and j. Suppose that $t_{s_j} > \mathrm{Tl}(j)$. This means that $\exists i'$ such that $t_{s_{i'}} + T_{i'} + C_{i',j} > \mathrm{Tl}(j)$. Then by induction hypothesis $\mathrm{Tl}(i') + T_{i'} + C_{i',j} > \mathrm{Tl}(j)$, which is not possible because $\mathrm{Tl}(j)$ is the longest path from node 1 to j. Hence, $t_{s_j} = \mathrm{Tl}(j)$. ∎

Proof of theorem 4.2.4 We shall prove the theorem by induction from the bottom of the graph. For the last task we have $\sigma_m = \sigma'_m = 0$. Suppose that it is true for all the successors of task i, then we prove that it is also true for task i. Let j^* be a successor of task i such that:

$$\begin{aligned} \sigma'_i &= S_{i \to j*} + \sigma'_j \\ &= S_{i \to j*} + \sigma_{j^*} \text{ (By hypothesis of induction)} \\ &= S_{i \to j^*} + M - \mathrm{Bl}(j^*) - \mathrm{Tl}(j^*) \end{aligned} \tag{4.4}$$

We call $\mathcal{P} = \{i, j, j+1, \ldots, m\}$ a path that goes from i to m but does not include the edge (i, j^*) and $\mathcal{P}^*$ is the path such that $\sum_{i \in \mathcal{P}^*} S_{i \to i+1}$ is minimum. We have $\mathcal{P}^* = \{i, j^*, j^*+1, \ldots, m\}$ and

$$\forall \mathcal{P} \sum_{j \in \mathcal{P}^*} S_{j \to j+1} \leq \sum_{j \in \mathcal{P}} S_{j \to j+1} \tag{4.5}$$

Let L be the length of the $\mathcal{P}$.

$$\begin{aligned}
L &= \sum_{j \in \mathcal{P}} (T_j + C_{j,j+1}) + T_m \\
&= t_{f_i} - t_{s_i} + C_{i,j} + t_{f_j} - t_{s_j} + C_{j,j+1} + \ldots + t_{f_{m-1}} - t_{s_{m-1}} + C_{m-1,m} + t_{f_m} - t_{s_m} \\
&= -t_{s_i} - S_{i \to j} - S_{j \to j+1} - \ldots - S_{m-1 \to m} + t_{f_m} \text{ (by definition of spare time)} \\
&= t_{f_m} - t_{s_i} - \sum_{j \in \mathcal{P}} S_{j \to j+1} \\
&\leq t_{f_m} - t_{s_i} - \sum_{j \in \mathcal{P}^*} S_{j \to j+1} \text{ (from inequality 4.5)} \\
&\leq \sum_{j \in \mathcal{P}^*} (T_j + C_{j,j+1}) + T_m
\end{aligned}$$

Hence the longest path that goes from i to m is the path where the sum of the spare time is maximum. In particular, it goes through j^*, which means that:

$$\mathrm{Bl}(i) = T_i + C_{i,j^*} + \mathrm{Bl}(j^*) \tag{4.6}$$

From Eq. 4.4 and 4.6, we then have:

$$\begin{aligned}
\sigma'_i &= S_{i \to j^*} + M - \mathrm{Bl}(i) + T_i + C_{i,j^*} - Tl(j^*) \\
&= M - Tl(j^*) + t_{s_{j^*}} - t_{f_i} - C_{i,j^*} + t_{f_i} - t_{s_i} + C_{i,j^*} \\
&= M - \mathrm{Bl}(i) - \mathrm{Tl}(j^*) - t_{s_i} + t_{s_{j^*}}
\end{aligned} \tag{4.7}$$

From Lemma 4.2.5, we have: $t_{j^*} = \mathrm{Tl}(j^*)$ and $t_i = \mathrm{Tl}(i)$, hence Eq. 4.7 becomes:

$$\sigma'_i = M - \mathrm{Bl}(i) - \mathrm{Tl}(i) = \sigma_i$$

■

In [16] the authors prove the following theorem:

Theorem 4.2.6. *Let i be a node with slack σ_i. If the duration of i exceeds its expected duration by $\Delta_i \leq \sigma_i$, then the makespan is unchanged, provided that all other nodes have a duration that does not exceed the expected duration.*

We can generalize the above theorem as follows:

Theorem 4.2.7. *Let i be a node with slack σ_i. If the duration of i exceeds its expected duration by $\Delta_i \leq \sigma_i$ then the makespan is unchanged, provided that all other nodes have a duration that does not exceed the expected duration. However, for all tasks j that are independent to task i in the disjunctive graph G', their own slack is unchanged.*

Proof of Theorem 4.2.7 For tasks i we have a new duration $T'_i = T_i + \sigma_i$. According to the proof of Theorem 4.2.6 (see [16]), the schedule is shifted this way: $\forall k \in \mathcal{P}_{i \to m}$, we just shift the start time of task k but not its duration such that $t_k = t_k + \delta_k$ and $t_{f_k} = t_{f_k} + \delta_k$ ($\delta_k = \max\left(0, \Delta_i - \sum_{l=i}^{k-1} S_{l \to l+1}\right)$). Let us show that the new slack σ'_j of task j is unchanged in the shifted schedule. Indeed, the lengths of the longest paths from task 1 to j and from j to m stay unchanged as i and j are independent (the durations of

the tasks on these two paths have not increased nor have the communications). Hence, $\mathrm{Bl}'(j) = \mathrm{Bl}(j)$, $\mathrm{Tl}'(j) = \mathrm{Tl}(j)$, and (from theorem 4.2.6) M is not increased by shifting the schedule, thus we have $\sigma'_j = M - \mathrm{Bl}'(j) - \mathrm{Tl}'(j) = M - \mathrm{Bl}(j) - \mathrm{Tl}(j) = \sigma_j$. ∎

This leads to the immediate following corollary:

Corollary 4.2.8. *If the expected time of several tasks is increased by a value smaller than their own slack and these tasks are all independent in the disjunctive graph, then the makespan is not increased.*

4.2.3 Robustness

A robust schedule is defined as a schedule that is insensitive to disturbances in task processing time. Robustness of a schedule provides a measurement of the degree of the "insensitiveness". In [66], the authors defined the robustness of a schedule as the linear combination of expected makespan and delay. This is one of the few early attempts to formalize the definition of schedule robustness. Unfortunately, the definition conflates the notion of robustness with the optimization criteria of makespan minimization, which limits its applicability. In [16], although the authors devised an empirical formula for robustness measure as an objective function to be optimized, the formula does not provide a way to evaluate the robustness of the schedule. We believe that the robustness of a schedule should reflect how stable the actual makespans will be with respect to the expected one. The overall performance of a schedule should consider both the expected makespan and the robustness. We propose two definitions in light of this perspective.

Definition 4.2.9. *Let $M_0(s)$ denote the expected makespan of schedule s obtained with expected task execution time and $M_i(s)$ the real makespan with i^{th} realization of expected task execution times. The relative schedule tardiness is:*

$$\delta_i(s) = \frac{\max(0, M_i(s) - M_0(s))}{M_0(s)} \tag{4.8}$$

The first definition of robustness of schedule s is:

$$R_1(s) = \frac{1}{E[\delta_i(s)]} \tag{4.9}$$

where $E[\cdot]$ represents the expectation operator.

Definition 4.2.10. *$M_0(s)$ and $M_i(s)$ are defined as above. N realizations of the expected task execution times are performed. Let $\mathcal{M} = \{M_i(s) | M_i(s) > M_0(s)\}$. The schedule miss rate is: $\alpha(s) = \frac{\|\mathcal{M}\|}{N}$. Then, the second definition of robustness of schedule s is:*

$$R_2(s) = \frac{1}{\alpha(s)} \tag{4.10}$$

4.3 A Bi-objective Task Scheduling Problem

As noted, there are two objectives in the context of robust task scheduling; minimizing the makespan and maximizing the robustness. In addition, we use average slack as the robustness measurement. The task of finding optimum solutions in this case is a *bi-objective optimization* problem. As will be shown in Section 4.4.2, these two objectives

are conflicting. Different solutions produce trade-offs between the two objectives, which means there is no single optimum solution. There exists a number of solutions that are all optimal. These solutions are called *non-dominated* solutions [32]. In dealing with such a bi-objective optimization problem, a few commonly used classical methods can be employed. In the next section, we will briefly describe the ϵ-constraint method [32] used in this study.

4.3.1 ϵ-constraint Method

ϵ-constraint method was proposed by Chankong and Haimes [23]. It is based on a scalarization where one of the objective functions is optimized while all other objective functions are bounded by some additional constraints. In the context of this study, the ϵ-constraint method can be formulated as follow:

$$\begin{cases} \text{Maximize} & \sigma \\ \text{Subject to} & M_0(s) < \epsilon \cdot M_{HEFT} \end{cases} \tag{4.11}$$

where σ is the average slack as defined in Eq. 4.3. ϵ is a user defined parameter. M_{HEFT} is the makespan of the schedule generated by the popular HEFT algorithm [98].

4.3.2 A Bi-objective Genetic Algorithm

We are now in a position to introduce the bi-objective genetic algorithm. The Genetic Algorithm (GA) is a powerful tool in finding global optimal solutions in large search

spaces. It has been used extensively in task scheduling [50, 100, 105]. There are many approaches to GAs in the literature. In this study, we implement a standard GA. In a standard GA, the first step is to encode any possible solution to the problem as a *chromosome*. Each chromosome represents a solution where a set of chromosomes is referred to as a *population*. Then an initial population is generated as the first generation from which the evolution starts. Each chromosome is associated with a *fitness value*, which represents the quality of the solution. The algorithm next *evaluate*s the quality of each chromosome with a problem-dependent *fitness function*. *Selection*, *crossover* and *mutation* are applied subsequently to the population to generate a population with better expected overall quality than the previous generation. These steps are repeated until the solution is converged according to predefined criteria. We present the details of each step of the bi-objective genetic algorithm below.

Chromosome representation

In GA, a chromosome representation, also called encoding, of a solution is a data structure that holds the information about the individual solution. In our GA based scheduling algorithm, each chromosome c_i consists of two parts; the *scheduling string* and *assignment strings*. The scheduling string is a topological sort of the task graph. This represents the execution order of the tasks. In a valid solution, the ordering of the task nodes in the scheduling string must observe the precedence constraints of the task graph. In the second part, p assignment strings represents the task assignment in each processor. Each string includes all tasks assigned to the processor that the string rep-

resents and the order of execution of the tasks on that processor. Including scheduling string in the chromosome can avoid illegal solutions where the precedence constraints are violated. In crossover and mutation steps, the scheduling string is used to enforce the precedence constraints among tasks.

Each generation of population contains a set of chromosomes. We denote the size of the population as N_p. In the GA, this size is kept constant throughout the evolution.

Initial population generation

Before the GA can evolve, an initial population must be generated. For each chromosome, a new scheduling string is produced by randomly generating a topological sort list. In forming the assignment strings, the algorithm chooses each task n_i from the newly created scheduling list in order and selects a processor p_j randomly. Then n_i is appended to the tail of string s_j, which represents the assignment string of p_j. As suggested in [100], it is a common practice in GA to incorporate solutions from some non-evolutionary heuristics into the initial population aiming to reduce the time needed for finding a near-optimal solution. In our GA, we include one chromosome that represents the solution from HEFT [98] in the population along with those generated randomly.

Newly generated chromosomes are checked for uniqueness. If a new chromosome is identical to any of the previously generated ones, it is discarded. Identical chromosomes could lead to a premature convergence where all chromosomes in a population have the same fitness values.

Fitness function

As noted, we use the ϵ-constraint method to solve the multi-objective optimization problem. In the GA, our goal is to maximize the average slack of the schedule subject to the constraint such that the makespan will not exceed some predefined threshold as formalized in Eq. 4.11. We can classify the individual solutions of the population into two categories; *feasible* ($\mathcal{F}$) and *infeasible* ($\mathcal{F}'$) solutions. Individuals in the first category satisfy the constraint in Eq. 4.11. Otherwise, they are categorized as infeasible solutions. The tenet of the ϵ-constraint method in dealing with MOOP is to choose one objective function as the only objective and the remaining objective functions as constraints. Therefore, those solutions that violate the constraint should be penalized in the fitness values. In light of this observation, the fitness of a chromosome c_i is set as follow:

$$fitness(c_i) = \begin{cases} \sigma & \text{if} \quad c_i \in \mathcal{F} \\ \min\{fitness(c_i) | c_i \in \mathcal{F}\} \cdot \frac{\epsilon \cdot M_{HEFT}}{M_0(c_i)} & \text{if} \quad c_i \in \mathcal{F}' \end{cases} \tag{4.12}$$

where σ, ϵ, M_{HEFT} and M_0 are defined the same as those in Eq. 4.11. For feasible solutions, the larger σ, the fitter. On the other hand, for infeasible solutions, those that severely violate the constraint are penalized more. Note that the above fitness function is population-based, where an individual chromosome's fitness is related to other chromosomes' fitness values.

Elitism is employed in the GA where the chromosome with the smallest fitness value

in the new population is replaced with the fittest chromosome in the current population. Elitism is an important mechanism that guarantees that the quality of the best solution found over generations is monotonically increasing.

Selection

The primary objective of the selection operator is to emphasize good solutions and eliminate bad ones in a population, while keeping the population size constant. It is designed to improve the average quality of the population by giving individuals of higher quality a higher probability to be copied into the next generation. There are several selection schemes proposed in the literature, such as *proportionate selection*, *ranking selection*, and *tournament selection* [48]. It has been shown that the tournament selection has better convergence and computational time complexity properties compared to any other selection operator that exists in the literature, when used in isolation. We implement the binary tournament in our GA. It works by choosing two individuals randomly from the population and copying the better one into the intermediate population. Then another two individuals are picked and the better one is put into the intermediate population. This process is repeated $2N_p$ times. Each individual can be made to participate in exactly two rounds of tournaments if done systematically. The best solution in a population will win both times, therefore making two copies of it in the new population. Similarly, the worst solution will lose in both tournaments and will be removed from the population. In this way, the average quality of the intermediate population is improved. The intermediate population is subject to crossover and mutation operators

to produce the next generation.

Crossover

In GA, crossover is an operator that combines the information of two individuals to produce one or two new individuals. The most common form of crossover involves two parents that produce two offspring. By exchanging parts of parent strings, usually starting from one or two randomly chosen crossover points, the offspring inherits desirable qualities from both parents. In this study, a single-point crossover is implemented. Two strings are chosen randomly as the parents to perform the crossover. First, a cutoff position is randomly selected. This divides the scheduling strings of both parents into two parts which we refer to as the *left* and *right* parts. Then the tasks in each right parts of the chromosomes are reordered to form the scheduling strings of the offspring. The left parts of the scheduling strings remain intact. The new ordering of the tasks in one right part is the relative positions of these tasks in the other parent's scheduling string. This guarantees that the newly generated scheduling strings are valid topological sortings of the task graph. Finally, for the assignment strings of the offspring, we first convert each parent's assignment string into a processor string representing each task's assigned processor number. Then, we randomly select a cut off point and exchange the right parts of the converted strings. Now the two new processor strings represent two new assignments. The offspring's assignment strings are formed by converting the processor strings back to their corresponding assignment strings.

In order to preserve some good strings selected during the selection operator, not

all strings in the population are used in crossover. If a crossover probability of p_c is used then $100p_c\%$ strings in the population are used in the crossover operation and $100(1 - p_c)\%$ of the population are simply copied to the new population.

Mutation

Mutation is GA's another way to explore the solution space. It can introduce traits not in the original population and keep the GA from converging prematurely before sampling the entire solution space. The classical mutation operator flips single bits in a string with a small mutation probability p_m. The mutation operator implemented in this GA works as follows. First, an individual is randomly chosen. Next, the mutation operator is applied to the selected chromosome with probability p_m. Then the mutation operator selects a task v randomly from the scheduling string and puts it in a new position such that the resulting new scheduling string does not violate the precedence constraint of the task graph thus guaranteeing the validity of the solution. This can be done by first identifying the *range* in which the select task can be place. The range is defined as the positions between the last position of the immediate predecessors of v and the first position of the immediate successors of v in the original scheduling string. Any position in the range is a valid choice. After task v is put into a new position in the scheduling string, a new processor p for v is picked at random. v is inserted into processor p's assignment string while maintaining the relative order of all the tasks assigned on that processor according to the scheduling string.

4.4 Experimental results and discussions

Our goal in the experiments is to answer the following questions: (1) Is slack an effective metric to control the robustness of a schedule? (2) How do the schedule's makespan and robustness change with respect to the ϵ value in the ϵ-constraint method used for solving the bi-objective optimization problem? (3) What is the best ϵ value when the overall performance that considers both the robustness and makespan is to be optimized?

In order to answer the above questions, extensive simulations have been carried out. Random task graphs are generated using the same method used in [92] with the following input parameters: task number n, shape parameter α, average computation cost ($\overline{cc}$), communication-to-computation ration (CCR). In the experiments, we set $n = 100$, $\alpha = 1.0$, $\overline{cc} = 20$ and $CCR = 0.1$. The best case execution time ($BCET$) matrix $\mathcal{B}$ is generated using the method suggested in [8]. It is a coefficient-of-variation(COV) based generation method. COV is a set of values that act as measures of heterogeneity. There are two different kinds of heterogeneity considered; *task heterogeneity* and *machine heterogeneity*. Task heterogeneity represents the degree to which the task execution times vary for a given machine. Similarly, machine heterogeneity is the degree to which the execution times vary for a given task. Four parameters, μ_{task}, V_{task}, μ_{mach}, V_{mach}, are defined in [8]. Among them, μ_{mach} can be obtained from the first two parameters. Thus, μ_{task}, V_{task},V_{mach} are three input parameters for the generation method. In fact, the average computation cost $\overline{cc}$ has the same definition as μ_{task}. We set $V_{task} = 0.5$ and $V_{mach} = 0.5$ to represent medium task and machine heterogeneities respectively.

Table 4.1: Values of the parameters used in the GA

Parameter	Description	Values
N_p	Number of chromosomes in the population	20
p_c	Crossover probability	0.9
p_m	Mutation probability	0.1

One important aspect of the experiments is to study how our algorithm will perform under different degrees of uncertainty in the actual resource environment. We use *uncertainty level* (UL) as a measurement of such degrees. Let $\mathrm{UL}_{i,j}$ be the uncertainty level of the execution time of task v_i on processor p_j, then the real execution time $c_{i,j}$ is a uniformly distributed random variable $\mathbf{U}(b_{i,j}, (2\mathrm{UL}_{i,j} - 1)b_{i,j})$, where $b_{i,j}$ is the best case execution time. So the expected execution time of v_i on p_j is $\mathrm{UL}_{i,j}b_{i,j}$. The $\mathrm{UL}_{i,j}$ matrix is generated similarly to the way we set the computation cost matrix. To start off, we have an average UL value for the graph. Then a vector $\mathbf{q} = \{q_1, q_2, ...q_n\}$, representing the expected uncertainty levels of each task, is generated according to gamma distribution $\mathbf{G}(1/V_1^2, \mathrm{UL} \cdot V_1^2)$. Finally, each $\mathrm{UL}_{i,j}$ is obtained according to gamma distribution $\mathbf{G}(1/V_2^2, q_i \cdot V_2^2)$. We set $V_1 = V_2 = 0.5$ in this study. The parameters of the GA are listed in Table 4.1.

The stopping criteria is that the number of iterations has reached 1000 or the current best solution has not improved over the last 100 iterations. Each experiment is repeated with 100 task graphs and for each task graph we have 1000 realizations of the expected task execution times.

4.4.1 Effectiveness of slack

In this section, we present our simulation results for studying the effectiveness of slack in increasing the robustness of the schedules. The results are shown in Fig. 4.2 and 4.3. Fig. 4.2 depicts the evolution process of a GA when the objective is to minimize the makespan. The solid lines represent makespan changes under different uncertainty levels. An initial observation is that when the uncertainty level is low, the GA can find schedules that have smaller makespans. For higher uncertainty levels, the GA fails to generate schedules with smaller makespans. Remember that when scheduling is performed, the GA only has the information about the expected task execution times. Each point forming the solid lines in Fig. 4.2 represents the makespan of the schedule generated by the GA when executed in the "real" environment with varying task execution time requirements. In fact, the expected makespan, which is the makespan of schedule when executed with the expected task execution times, is decreasing during the evolution process. For a high uncertainty level, the GA tends to "overfit" the schedule based on the expected task execution times, which leads to an increasing makespan in the real resource environment. Fig. 4.2 also shows that when minimizing the makespan is the goal of the GA, schedules will have smaller slacks and robustness with the advance of the stages of evolution process. This is due to the fact that a schedule with a small makespan tends to leave a small time "window" for each task, thus resulting in small slack. For a low uncertainty level, the decrease of slack and robustness is more significant because GA finds schedule with considerably smaller makespan at such case.

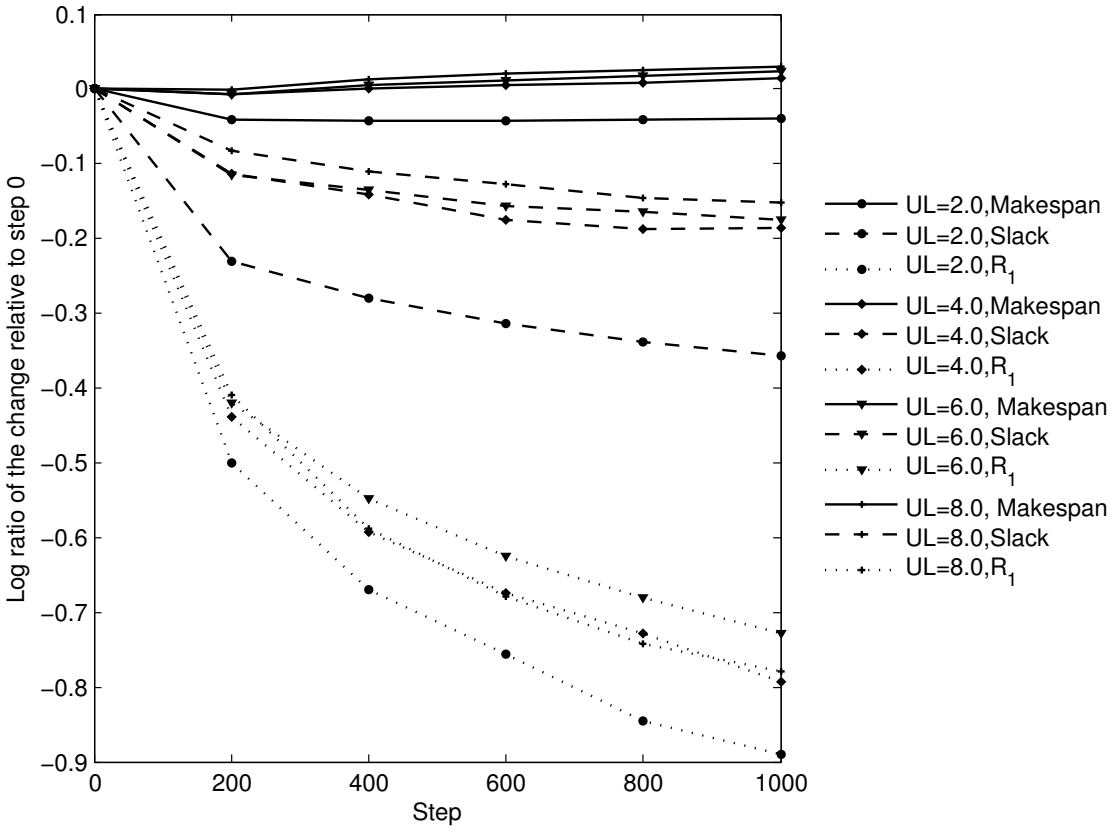


Figure 4.2: Evolution of a GA when minimizing the makespan is the objective function

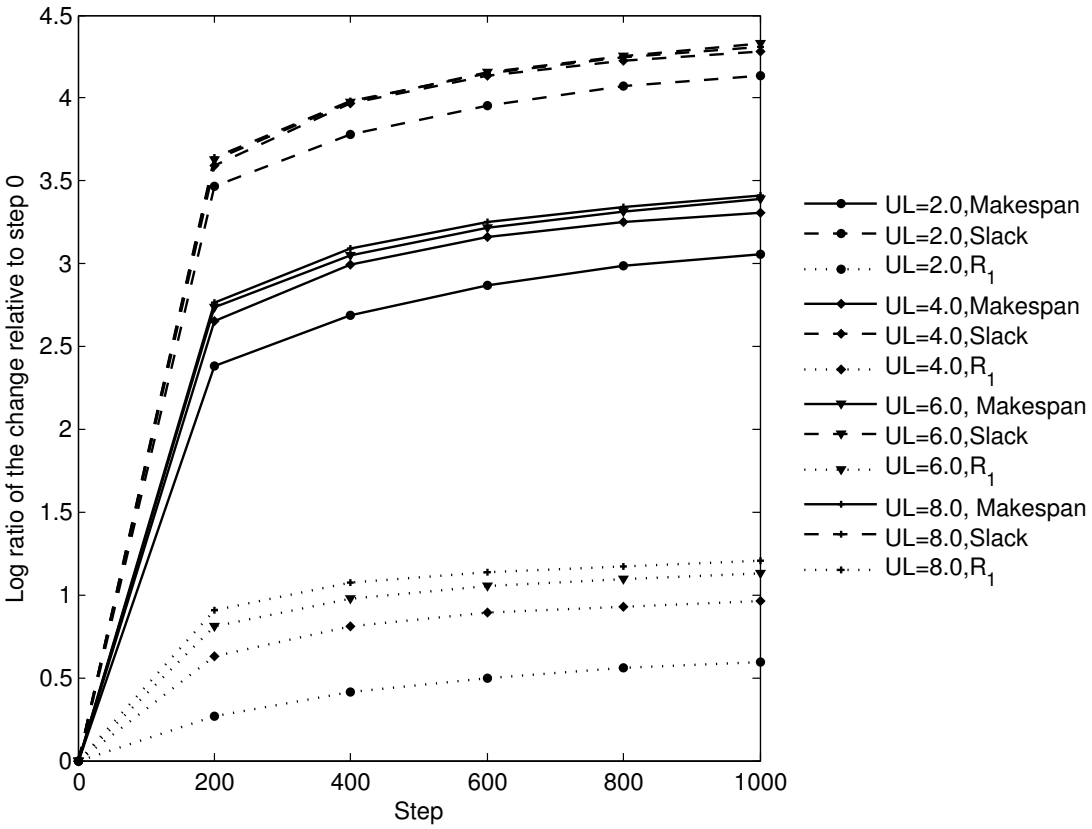


Figure 4.3: Evolution of a GA when maximizing the slack is the objective function

Fig. 4.3 presents the evolvement of makespan, slack and robustness when the GA's goal is to maximizing the slack of the schedule. It can be observed that with the increase of slack, the robustness also improves. At the same time the makespan rises substantially.

From Fig. 4.2 and 4.3, we conclude that the slack is an effective metric that can be used to increase the robustness of a schedule. The goals of maximizing the slack and minimizing the makespan are conflicting. We present the results of the bi-objective optimization problem in the next sections.

4.4.2 Results of solving the bi-objective optimization problem

In this section, we show the results of solving the bi-objective optimization problem using the ϵ-constraint method. First, we let $\epsilon = 1.0$, which means that during the evolution, only those schedules with expected makespan less or equal to the makespan of the schedule obtained with HEFT are feasible schedules. Infeasible schedules always have fitness values smaller than any feasible schedule. Fig. 4.4 shows the log-ratio of relative improvement of several performance metrics over those of schedules generated by the HEFT algorithm. We observe the following from this figure: (1) the average makespan of the schedules obtained with the GA algorithm still outperforms that of the schedules generated by the HEFT algorithm, especially when the uncertainty level is not very high. Remember that the main purpose of this experiment is to maximize the robustness while restricting the makespan not to exceed that of schedules obtained by HEFT. (2) the figure clearly indicates that robustness based on tardiness (R_1) improved significantly.

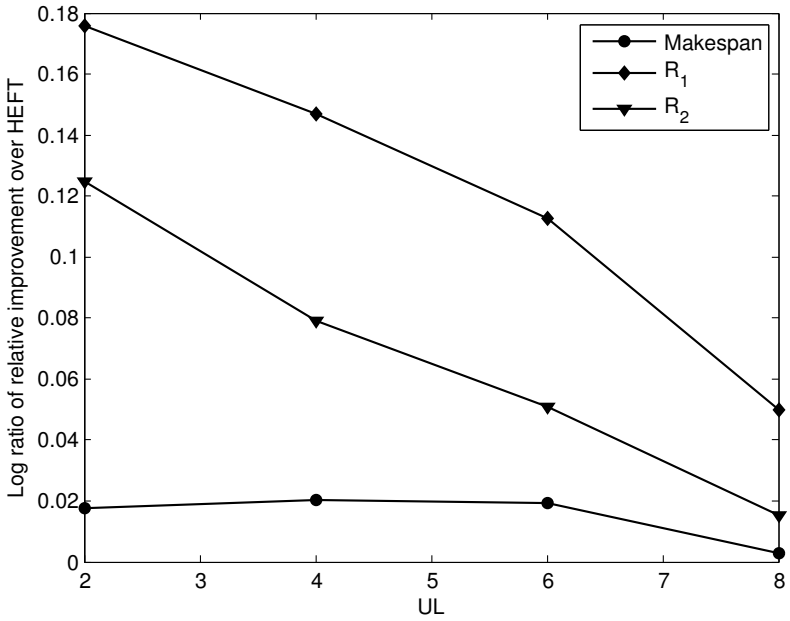


Figure 4.4: Performance improvement over HEFT ($\epsilon = 1.0$)

For example, at UL = 2, the robustness is increased by 13%. The improvement is less significant at high uncertainty levels. This is due to the fact that at high uncertainty levels, the increased slack, which is not much because we limit the makespan increase, is not sufficient to absorb the uncertainty, thus limiting the improvement of robustness. (3) Similar observations can be made for robustness based on miss rate (R_2). The improvement is less considerable compared with that of R_1.

Because limiting the ϵ value also limits the chance of robustness improvement, especially when uncertainty level is high as shown above, we next investigate how the robustness can be improved by relaxing the ϵ requirement. Fig. 4.5 and 4.6 show the comparison of the improvement of R_1 and R_2 at various uncertainty levels to the im-

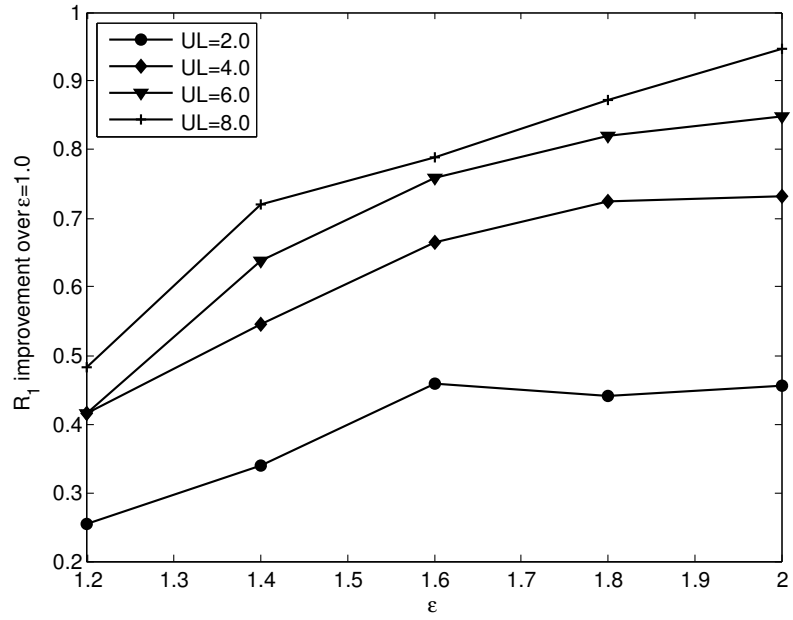


Figure 4.5: R_1 improvement over $\epsilon = 1.0$

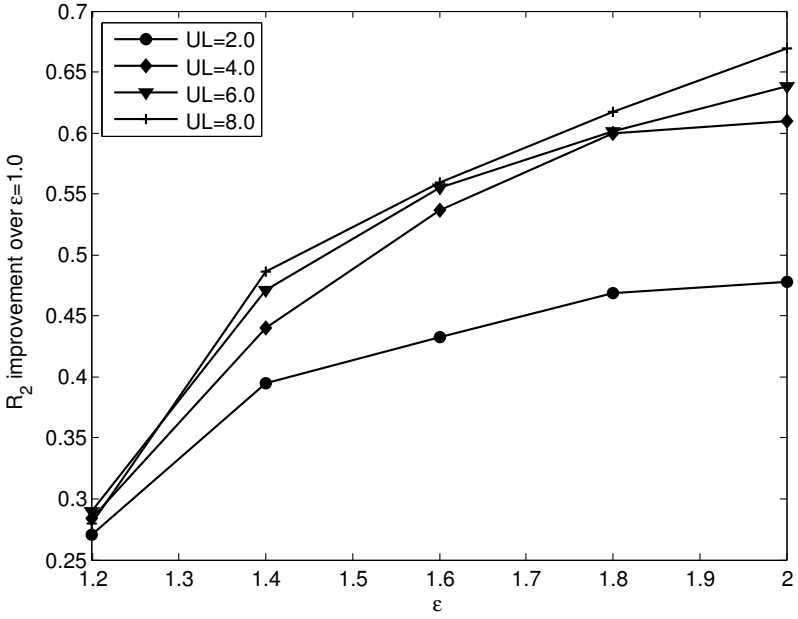


Figure 4.6: R_2 improvement over $\epsilon = 1.0$

provement when $\epsilon = 1.0$. The y-axes are log-scaled. As can be seen from the figures, with the increase of ϵ value, there will be more slack to absorb the uncertainty, thus improving the robustness of the schedules. Also we observe that for high uncertainty level, the relative improvement is larger and is leveled at larger ϵ values. This can be explained by noticing that at high uncertainty levels there is more "room" for improvement, so increasing ϵ can be very effective. For example, at UL = 2.0, there is relatively no more improvement of R_1 after $\epsilon = 1.6$. By contrast, at UL = 8.0, the robustness is still improving when $\epsilon = 2.0$. We can make another observation by comparing Fig. 4.5 and 4.6: the improvements of R_2 at different uncertainty levels is not as disparate as those of R_1. It suggests that R_2 is less sensitive to uncertainty level as R_1.

Since makespan and robustness are two important metrics in evaluating a schedule and are conflicting with each other, we propose using the following weighted sum of the two metrics as a means to represent the overall performance of schedule s.

$$P(s) = r \log \frac{M_{HEFT}}{M(S)} + (1 - r) \log \frac{R(s)}{R_{HEFT}} \tag{4.13}$$

where M_{HEFT}, R_{HEFT} is the makespan and robustness of the schedule obtained by the HEFT algorithm, respectively. r $(0 \leq r \leq 1)$ is a weight given by the user. If the user puts more emphasis on having a small makespan, a large r should be applied. Otherwise, if the user prefers a schedule with relatively large robustness, then r should be set to a number close to 0.

Fig. 4.7 and 4.8 show the values of ϵ $(1.0 \leq \epsilon \leq 2.0)$ when the best overall perfor-

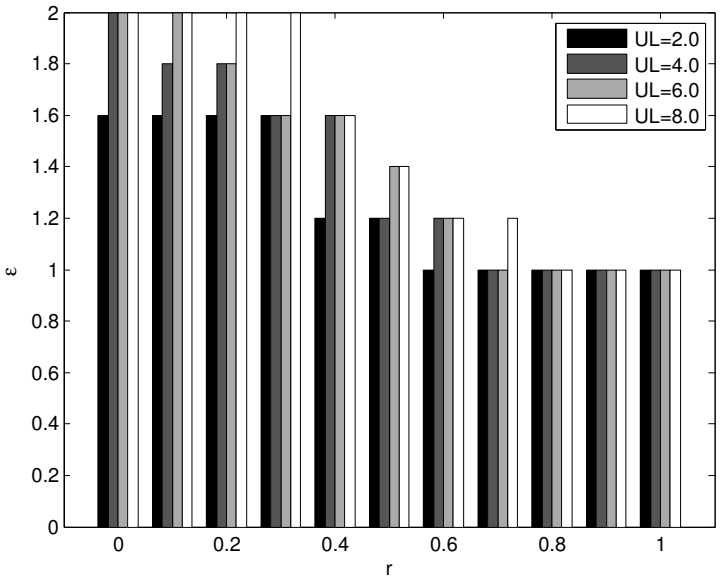


Figure 4.7: The best ϵ value for overall performance based on R_1 and makespan

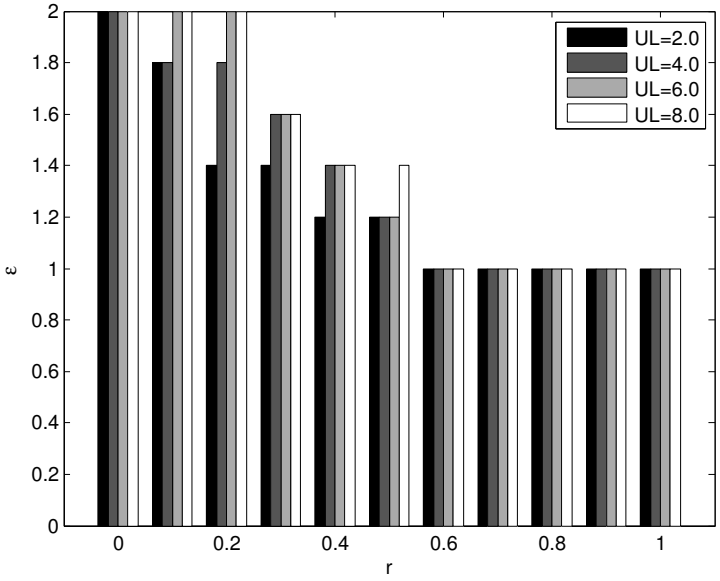


Figure 4.8: The best ϵ value for overall performance based on R_2 and makespan

mance with different r is achieved. We use R_1 (resp. R_2) in the definition of overall performance (Eq. 4.13 in Fig. 4.7 (resp. Fig. 4.8)). With the increase of r, we put more emphasis on the makespan. The figures clearly indicate that in order to achieve the best overall performance with large r, a small ϵ value should be used. On the other hand, if a schedule with large robustness is desired (r is small), then a large ϵ value is preferable. Furthermore, for higher uncertainty level, a larger ϵ is required in order to obtain better overall performance.

4.5 Conclusions

In this chapter, we developed an algorithm for matching and scheduling of DAG-structured applications with the goals of both minimizing the makespan and maximizing the robustness. Due to the fact that the two goals are conflicting, satisfying both objectives at the same time is usually impossible. We used the ϵ-constraint method to solve the bi-objective optimization problem. We proved that slack is an effective metric to be used to adjust the robustness, and it is confirmed subsequently that slack and robustness are positively related. Two definitions of robustness based on tardiness and miss rate were proposed. Experiments showed that considering the slack as an objective can greatly improve the robustness while we confined the makespan not to exceed that of HEFT. By relaxing the requirement of makespan, the robustness can be improved furthermore. The algorithm is found to be flexible to find the ϵ value in a certain user provided range so that the best overall performance considering both makespan and

robustness is achieved.

Chapter 5

Stochastic Task Scheduling in HDCS

5.1 Introduction

In today's distributed systems, the computing environments are inherently heterogeneous both spatially and temporally. The computation and communication resources in a system are usually shared by multiple users. For example, the emerging Grid platforms [41] integrate and coordinate resources and users that are located within different control domains. Resource sharing is essential and is one of the most important features of Grids. Grids are inherently large, heterogeneous and dynamic systems. In such dynamic systems, environmental characteristics such as available CPU, network bandwidth, etc., are likely to vary. In [90], the authors used *point value* to refer to the

single value that is given to each characteristics for the performance prediction model. They argued that it is more useful to represent the characteristics as a *distribution* of possible values over a range instead of a point value. Such a value is called the *stochastic value*. Two representations of stochastic value are used; *normal distribution* and *interval*. Experiments show their effectiveness in predicting the performance of several parallel applications under different workloads on a shared network of stations. In [89], the authors showed that performance predictions with stochastic values can be used effectively by the scheduler to improve the performance of the application in a similar environment. The stochastic scheduling policy is based on time balancing for data parallel applications. It is modified to use stochastic information for determining the data allocation. Experiment results demonstrate that it is possible to achieve faster application execution times and more predictable application behavior using stochastic scheduling.

In this chapter, we investigate the problem of scheduling a task graph using stochastic information of the system performance. In section 5.2, we formulate the stochastic DAG scheduling problem. We present a method for calculating the makespan distribution of a task graph in section 5.3. Section 5.4 describes experiment settings. We provide our simulation results and discussions in section 5.5. Finally, section 5.6 concludes this chapter.

5.2 Stochastic DAG scheduling problem

We refer to the problem of DAG scheduling with stochastic environmental characteristics information simply as *stochastic scheduling*. The stochastic scheduling problem is formulated in this section. As before, we will consider the problem of scheduling a task graph $\mathcal{G}$ of size n onto a set of m heterogeneous machines $\mathcal{P}$. The execution time of task i on machine j is represented by random variable $\tau_{i,j}$. Similarly, the data transfer speed between processors i and j is represented by $c_{i,j}$ (unit of data/unit time).

Suppose that a schedule s is given. Let $T_S(s)$ denote the makespan in the stochastic scheduling problem with schedule s. Note that $T_S(s)$ is a probabilistic distribution of the makespan. The stochastic scheduling problem can be formulated as:

$$\min_{s\in\Pi}\{E[T_S(s)]\} \tag{5.1}$$

where $E[\cdot]$ is the expectation operator and Π represents all valid schedules for the problem. On the other hand, the deterministic scheduling problem, where all the characteristics are represented by their expected values, can be defined as:

$$\min_{s\in\Pi}\{T_D(s)\} \tag{5.2}$$

where $T_D(s)$ is the makespan under schedule s.

We are now in a position to present a genetic algorithm for the stochastic scheduling problem. We refer to it as the *stochastic DAG scheduling* algorithm (SDS). In com-

parison, *deterministic DAG scheduling* algorithm (DDS) deals with the deterministic scheduling problem. The basic elements of the genetic algorithm include the fitness function for a chromosome, the selection, crossover and mutation operators. Except for the fitness function, we adopt the same selection, crossover and mutation operators as those described in Chapter 4.

The fitness function of a chromosome is used to evaluate the quality of the potential solution. It is always problem dependent. Even for the same problem, several fitness functions can be devised to evaluate the quality of the different aspects of the solutions. One of the goals of the work presented in this chapter is to compare the schedules obtained with SDS with those obtained using DDS. The only difference between SDS and DDS is the fitness function employed. In the DDS algorithm, the final makespan of a schedule is a point value. Therefore, the fitness function for a schedule is the makespan for that schedule. Given a chromosome c, the fitness of c, which is denoted by $f(c)$, is determined by:

$$f(c) = T_D(c) \tag{5.3}$$

However, in the SDS algorithm, given a schedule, the makespan corresponding to that schedule is a stochastic value that provides more useful information about the makespan than a point value. There are several possible choices for the fitness function in this case. The most natural one is to take the expectation of the stochastic makespan value

as the fitness function. In this case, the fitness value of c is:

$$f(c) = E[T_S(c)] \tag{5.4}$$

Such a choice is based on the fact that expectation of a random variable X gives an idea of "what to expect" from the repeated outcomes of the experiment represented by X. As demonstrated in Chapter 4, another important aspect of the quality of a schedule is the robustness. In probability theory, the *Coefficient of Variation* (COV) of a random variable X is used to measure the dispersion of the X's probability distribution. It is defined as the ratio of the standard deviation to the mean:

$$COV(X) = \frac{\sigma}{\mu} \tag{5.5}$$

where σ is the standard deviation of X and μ the mean of X. COV is closely related to the robustness of the schedule. A large COV typically means that the schedule has a makespan distribution with a small dispersion, thus more robust. It was shown in Chapter 4 that the robustness and makespan of a schedule are two conflicting objectives. Therefore, we employ a bi-objective algorithm to take into account both the makespan and robustness of the schedule. Similar to the setting of fitness function presented in Chapter 4, the fitness value of a chromosome c is the aggregation of two objective

functions. It can be formulated as:

$$f(c) = (1-w)\frac{f^1(c) - f^1_{min}}{f^1_{max} - f^1_{min}} + w\frac{f^2(c) - f^2_{min}}{f^2_{max} - f^2_{min}} \tag{5.6}$$

where $f^i(c)$ $(i = 1, 2)$ is the fitness value corresponding to the ith objective. f^i_{max} and f^i_{min}, $(i = 1, 2)$ are the best and worst fitness values found for the ith objective. In our algorithm, we identify the first and second objective as $E[T_S(c)]$ and $COV(T_S(c))$, respectively. w is a user defined weight between 0 and 1.

5.3 Calculation of the makespan distribution of a task graph

5.3.1 Estimating the makespan distribution

While it is easy to calculate the makespan of a task graph with a given schedule when the environmental characteristics are point values, it is extremely difficult to obtain the makespan distribution in the case of stochastic values as demonstrated below. The difficulty is due to two basic operations involving random variables; *addition* and *maximum*. From probability theory, if two independent random variables X and Y with probability density functions (PDFs) given by $f_X(t)$ and $f_Y(t)$ and Z is a random variable that is the sum of X and Y, then the following holds:

$$Z = X + Y \tag{5.7}$$

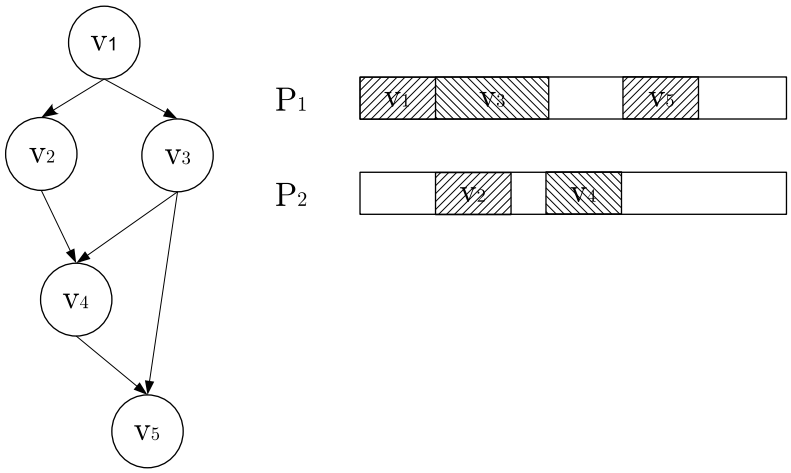


Figure 5.1: A task graph and an assignment on two processors

$$f_Z(t) = f_X(t) \otimes f_Y(t) = \int_{-\infty}^{\infty} f_X(u) f_Y(t-u) du \tag{5.8}$$

$$E[Z] = E[X] + E[Y] \tag{5.9}$$

where $\otimes$ is the convolution operator. Let W be the maximum of X and Y. $F_X(t)$ and $F_Y(t)$ are the cumulative distribution functions (CDFs) of X and Y, respectively.

$$W = \max(X, Y) \tag{5.10}$$

$$F_W(t) = F_X(t) \cdot F_Y(t) \tag{5.11}$$

$$E[W] > \max(E[X], E[Y]) \tag{5.12}$$

Consider the task graph with five nodes shown in Fig. 5.1. The five tasks are assigned on two processors as illustrated in the right part of Fig. 5.1. It is assumed that there is

no communication cost between the two processors for the sake of a clearer presentation. We use $ST(i)$ to denote the start time of task i and $FT(i)$ the finish time of i. Recall that $\tau_{i,j}$ is the execution time of task i on processor j. It is easy to derive step by step the makespan of the schedule as follows:

$$FT(1) = \tau_{1,1}$$

$$ST(2) = FT(1) = \tau_{1,1}$$

$$ST(3) = FT(1) = \tau_{1,1}$$

$$FT(2) = ST(2) + \tau_{2,2} = \tau_{1,1} + \tau_{2,2}$$

$$FT(3) = ST(3) + \tau_{3,1} = \tau_{1,1} + \tau_{3,1}$$

$$\begin{aligned} ST(4) &= \max\{FT(2), FT(3)\} \\ &= \max\{\tau_{1,1} + \tau_{2,2}, \tau_{1,1} + \tau_{3,1}\} \end{aligned}$$

$$\begin{aligned} FT(4) &= ST(4) + \tau_{4,2} \\ &\max\{\tau_{1,1} + \tau_{2,2}, \tau_{1,1} + \tau_{3,1}\} + \tau_{4,2} \end{aligned}$$

$$ST(5) = \max\{FT(3), FT(4)\}$$

$$makespan = FT(5) = ST(5) + \tau_{5,1}$$

Due to dependencies, Eqs. 5.8 and 5.11 can not be applied to calculate the addition and

maximum of two random variables. For example, $FT(2)$ and $FT(3)$ are not independent, Eq. 5.11 is not applicable to obtain $ST(4)$. An exact derivation of the final makespan distribution using probability theory becomes extremely complex and involves multiple integrals [67].

In order to simplify the scheduling processor, we need to find an easy way to estimate the makespan distribution without a great loss of accuracy. As noted previously, the complexity of calculating the distribution originates from the fact that random variables are dependent on each other. Tasks with a common ancestor can have correlated random variables associated with the start and finish times. As suggested in [67], the Kleinrock Independence Approximation (KIA) [60] can be used as a basis for assuming independence among random variables that may be correlated in this case. KIA was originally proposed in the context of data networks. It provides an approach to analyze delay in networks carrying multiple packet streams. In a typical data network, there are multiple interacting transmission queues. A packet stream enters the network at the source node's queue then joins one or more other queues before leaving the network at the destination node's queue. KIA states that although the packet inter-arrival times can become dependent after the stream leaves the first queue, the merging of many different packet streams on a transmission line restores the independence of inter-arrival times and service times. In the case of a task graph, we can view the data dependency among the tasks as the merging of data streams. Thus according to KIA, the input data (streams) from many other predecessors has the effect of restoring independence among

the start and finish times of tasks that have a common ancestor in the graph. This will indeed facilitate the calculation of the makespan distribution since we can now use Eqs. 5.8 and 5.11 to compute the start and finish times of each task. In section 5.5.1, we will investigate the accuracy of the estimation with respect to the characteristics of the resource environment and the task graph.

The calculation of the makespan distribution with the assumption of KIA can be carried out with the following steps:

1. Based on the given schedule, create the disjunctive graph, which includes the task and machine dependencies.

2. For each task v_i, compute the start time $ST(i)$. The CDF of $ST(i)$ is obtained with:

$$F_{ST(i)}(t) = \prod_{j=0}^{pred(i)-1} F_{FT(j)}(t) \tag{5.13}$$

   where $pred(i)$ is the number of immediate predecessors of v_i.

3. For each task v_i, let $s(i)$ denote the machine where task v_i is assgined. The PDF of the completion time of v_i is:

$$f_{FT(i)}(t) = f_{ST(i)}(t) \otimes f_{\tau_{i,s(i)}}(t) \tag{5.14}$$

4. If we have n_e exit nodes in the graph, namely, v_{e_0},v_{e_1},...,$v_{e_{n_e-1}}$. then the CDF of

the makespan distribution is:

$$F(t) = \prod_{k=0}^{n_e-1} F_{FT(e_k)}(t) \tag{5.15}$$

5.3.2 Numerical calculation of maximum and addition of two random variables

Random variables with certain probability distributions are used to represent the numerical outcomes of a random phenomenon. The problem of determining the distributions of random variables whose samples are functions of samples of other random variables is very common and has received much attention. In [96], a survey of theoretical approaches to the basic algebra operations of random variables was presented. However, analytical methods only apply to certain classes of distributions, such as normal, exponential, etc. Numerical method tends to be applicable to a much wider class of distributions. There are basically two broad categories of approaches for the numerical methods. Monte-Carlo is the traditional method with certain serious drawbacks [39], such as difficulties in handling random variables that have unknown dependencies or that have imprecise probabilities, i.e., with distributions that are not fully specified. Non-Monte Carlo methods involve discretizing of probabilistic distribution followed by computation on the discretized forms. They can be further divided into two groups. In the earlier algorithms, it was assumed that the random variables are independent [24, 52, 57, 72]. Later, approaches based on the theory of copulas [76] were studied. These approaches

focused on finding the bounds for joint distributions from their given marginal distributions in the presence of unknown dependency relationships among the random variables [102, 103]. Berleant et al. developed DEnv algorithm based on the discretization of random variables [10, 11, 12, 13]. It uses linear programming to achieve dependency bounds for random variables that may be independent, have unknown dependency or have a dependency with partial information. Upper and lower bounds on the CDF are calculated by integrating across the left side, top and bottom, and right side of the histograms. Then they are combined to form an upper and lower CDF that is guaranteed to be valid.

Since we are dealing with operations of independent random variables (recall that using KIA, we assume that the start and finish times of tasks with a common ancestor are independent), we will use the *histogram method* [52] to compute operations on PDFs, specifically, the addition and maximum. The histogram method first discretizes PDF operands using *intervals*. It then uses interval operations to generate intermediate results followed by constructing the final PDF from the intermediate results. In the histogram method, PDF operands are discretized using histograms. A histogram is defined as a bar graph that shows frequency data. Each bar is characterized both by an interval describing its placement on the real number line and by a probability mass associated with that interval. Consider the problem of calculating $Z = X \square Y$, where X and Y are two random variables with PDFs $f_X(t)$ and $f_Y(t)$. $\square$ represents an operator such as '+', '-', 'max', etc. We first discretize $f_X(t)$ and $f_Y(t)$ over $[x_a, x_b]$ and $[y_a, y_b]$

respectively. Let n_X and n_Y denote the number of *bin*s for X and Y. Then, the histogram H_X and H_Y for X and Y can be represented by a collection of 2-tuples.

$$H_X : \{([x_i, x_{i+1}], p_i) | 0 \leq i \leq n_X - 1, x_0 = x_a, x_{n_X - 1} = x_b\} \tag{5.16}$$

$$H_Y : \{([y_i, y_{i+1}], q_i) | 0 \leq i \leq n_Y - 1, y_0 = y_a, y_{n_Y - 1} = y_b\} \tag{5.17}$$

where

$$p_i = \int_{x_i}^{x_{i+1}} f_X(t)dt \tag{5.18}$$

$$q_i = \int_{y_i}^{y_{i+1}} f_Y(t)dt \tag{5.19}$$

is the probability mass of the corresponding interval. We define a function $Prob$:

$$Prob_H(I) = p \tag{5.20}$$

where I is an interval of histogram H and p is the probability mass for that interval. For those distributions with infinite support, we discretize it over the range where the total probability mass is higher than some threshold (e.g. > 0.99). Then a normalization step is performed so that the resulting probability mass is summed to 1. After the PDFs are discretized, they are combined as follows:

1. Perform a Cartesian product of the intervals of the histograms describing X and

Y:

$$\Phi = \{(I_s, J_t) | I_s \in \{[x_i, x_{i+1}], 0 \leq i \leq n_X - 1\}, J_t \in \{[y_i, y_{i+1}], 0 \leq i \leq n_Y - 1\}\} \tag{5.21}$$

2. For each element in Φ, we produce an intermediate histogram bar in the following fashion:

   (i) The new interval is obtained by performing a corresponding interval operation [71] on I_s and J_t to get $K_{st} = \square(I_s, J_t)$. For example, if we are calculating the maximum of X and Y, then $K_{st} = \max\{I_s, J_t\}$ is performed.

   (ii) The probability mass of the new interval K_{st} is

$$Prob(K_{st}) = Prob(I_s) \cdot Prob(J_t) \tag{5.22}$$

3. The intermediate histogram is combined to get the final PDF.

   (i) First, we discretize the support of Z into n_Z bins. The number of bins (we will use b to represent it hereafter) is related to the accuracy of the final result. The larger n_Z, the more accurate the result will be. We denote all the bins (intervals) of Z as $B_Z = \{[z_i, z_{i+1}] | 0 \leq i \leq n_Z - 1\}$.

   (ii) Calculate the probability mass for each interval of Z defined in the previous step as follows: If the intermediate interval K_{st} completely falls within some element of B_Z, then $Prob(K_{st})$ is assigned to that element. For example,

if $\exists i$, such that $K_{st} \subseteq [z_i, z_{i+1}]$, then $Prob([z_i, z_{i+1}])+=Prob(K_{st})$. If K_{st} covers more than one element of the B_Z, then its probability mass is shared by those elements proportional to the fraction of the interval it covers. This is called *proportional assignment*. Then all the probability mass assigned to each element of B_Z is summed to be the total probability mass for that bin.

5.4 Simulation

Our goals in the simulation experiments include:

1. to study the accuracy of the method laid out in Sec. 5.3 in approximating the makespan distribution in the case of stochastic scheduling,
2. to investigate whether scheduling with stochastic information can lead to a better schedule in terms of minimizing the schedule length and maximizing the robustness of the schedule, and
3. to evaluate the effectiveness of the bi-objective algorithm for schedule length and robustness trade-off.

The task graphs used in the experiments are generated randomly with the following parameters:

- The number of tasks in the graph (n).
- The height parameter of the graph (α). The height of a graph (h) is determined by $h = \sqrt{n}/\alpha$.

- Average in-degree of the nodes (d). The in-degree of a node is randomly generated from a uniform distribution with mean value d.

- Average computation cost ($\overline{c}$). The average computation cost of task i on processor j, $\overline{c}_{i,j}$ is generated randomly from a uniform distribution with mean value $\overline{c}$.

- Communication-to-Computation Ratio (CCR). This is the ratio of average communication cost to the average computation cost.

- Average communication cost ($\overline{r}$). This is obtained with

$$\overline{r} = CCR \cdot \overline{c} \tag{5.23}$$

- The average scale parameter (θ). We use gamma distribution to model the execution time and data transfer rate. Gamma distributions are used for simulating the variables due to the fact that with proper settings of their characteristic parameters, they can approximate other popular probability distributions, such as the Erlang-k and Gaussian (without the negative values) distributions. This is helpful because the simulated random variables can be synthesized closer to some real life heterogeneous computing systems. Recall that $\tau_{i,j}$ represents the execution time of task i on processor j. Then,

$$\tau_{i,j} \sim Gamma(k_{i,j}, \theta_{i,j}) \tag{5.24}$$

Table 5.1: Parameters used in the simulations

Parameter	Description	Values
n	number of tasks	$20, 40, 60, 80, 100$
α	height parameter of the graph	$0.5, 1.0, 2.0$
d	average in-degree of the nodes	$2.0, 3.0, 4.0, 5.0, 6.0$
$\overline{c}$	average computation cost	$20, 40, 60$
CCR	communication-to-computation ratio	$0.1, 1.0, 10.0$
θ	average scale parameter	$1.0, 2.0, 3.0, 4.0, 5.0$

where $k_{i,j} > 0$ is the *shape parameter* and $\theta_{i,j} > 0$ is the *scale parameter*. The expected computation cost of task i on processor j, $\overline{c}_{i,j}$, $k_{i,j}$ and $\theta_{i,j}$ is governed by:

$$\overline{c}_{i,j} = k_{i,j} \cdot \theta_{i,j} \tag{5.25}$$

The scale parameter $\theta_{i,j}$ is randomly generated from a uniform distribution with mean value θ. The scale parameter is closely related to the dispersion of the gamma distribution. Large θ means the distribution is more spread out, i.e., with large dispersion. The above reasoning also applies to the communication cost (data transfer rate).

The values of the parameters used in the simulations are summarized in Table 5.1. The processor number used in the experiments is 16. The parameters for the genetic algorithms used are the same as those used in Chapter 4. (See Table 4.1).

In the first set of experiments, we investigate how several characteristics of the task graph and the approximation method affect the accuracy in estimating the makespan

distribution. These characteristics include the size of the graph (n) and the bin number (b) used in the approximation method. The estimations obtained using approximation method described in Sec. 5.3 and those with the Monte-Carlo method are compared. For each set of parameters used, 50 different schedules is generated. The average makespan over these repetitions is then compared. In Monte-Carlo simulation, 1000 repetitions are performed for each schedule. The second part of the simulations performs the comparison between SDS and DDS. During the comparison, we first generate the task graph with different sets of parameters. Then each algorithm is applied to obtain a schedule. Finally, the schedules are examined with simulated actual execution time and data transfer rate. The actual makespan and robustness are compared. As noted, the makespan and robustness of a schedule are two conflicting factors. In the last part of the experiment, we study the effectiveness of weight w in trading off between the two factors.

5.5 Results and discussion

5.5.1 Accuracy of the estimation

In this section, the accuracy of the makespan distributions obtained from the approach described in Sec. 5.3 is evaluated. We compare the distributions with those determined from the Monte-Carlo simulations due to the fact that derivation of the exact distribution for the task graphs considered in the experiments is not feasible.

For each instance of the Monte-Carlo simulation, the execution time and data trans-

fer rate is determined by generating a random number according to their assumed probability distribution, i.e., gamma distribution with specified parameters. The overall makespan of the task graph can be calculated once a schedule is given. For each graph studied, 1000 simulation instances are performed to produce the sample distribution of the overall makespan of the task graph.

The experimental results of studying the accuracy of the makespan distribution estimation approach are given in Figs. 5.2 – 5.9. Figs. 5.2 and 5.3 shows the relative deviation of the estimated makespan distribution's expected value and standard deviation from those of the distribution obtained with Monte-Carlo simulations when the task graph size n varies ($b = 2000, \theta = 1.0, d = 3$). In general, it can be observed that, as the task graph size increases, the relative deviation also increases. Large size task graphs tend to have more complicated data dependency structures provided that the average in-degree of the nodes is kept the same. During the estimation of the makespan distribution, the error of random variable operations accumulates along the critical path. Graph with large number of task nodes usually has longer critical path. In such a case, the accumulated error of random variable operations is more significant. We also calculate the makespan based on the expected values of task execution time and data transfer rate. It is evident that the makespans obtained with expected values is less accurate than those obtained with estimation using random variables. Fig. 5.2 shows that with the increase of task graph size, the difference becomes more considerable.

As noted in Sec. 5.3, the bin number b used in the discretization of a probabilistic

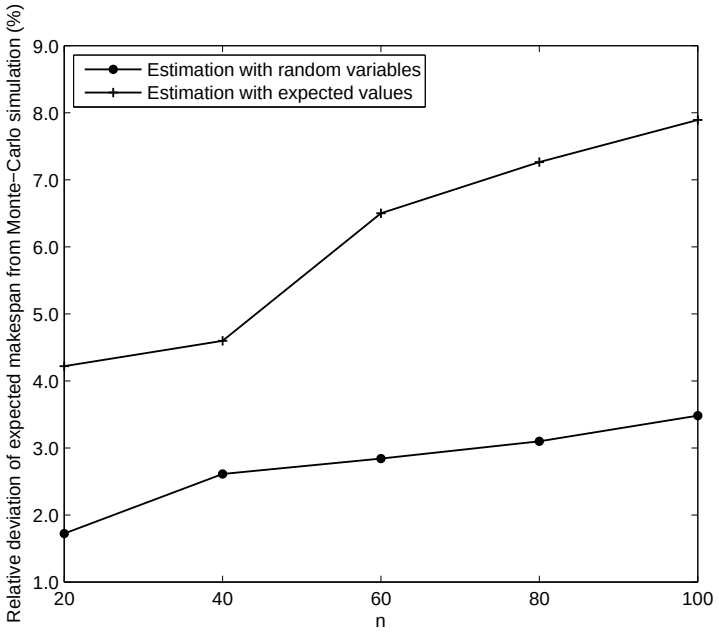


Figure 5.2: Relative deviation of expected makespan from Monte-Carlo simulation for graphs of different sizes

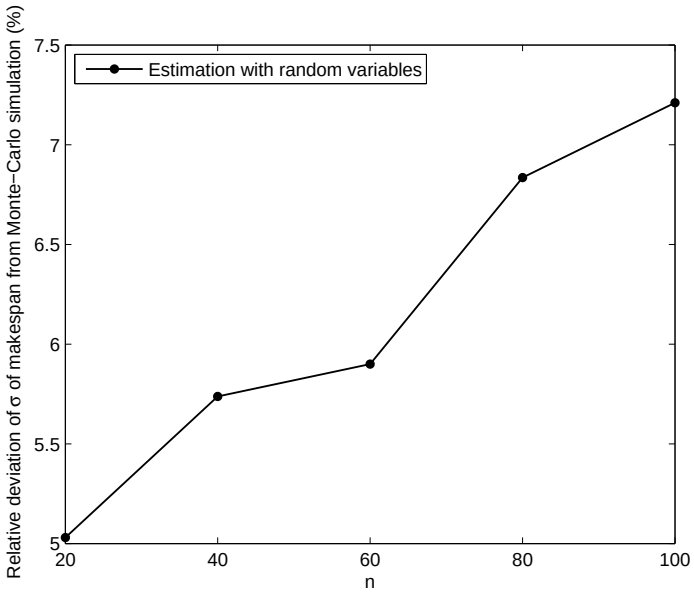


Figure 5.3: Relative deviation of σ of makespan from Monte-Carlo simulation for graphs of different sizes

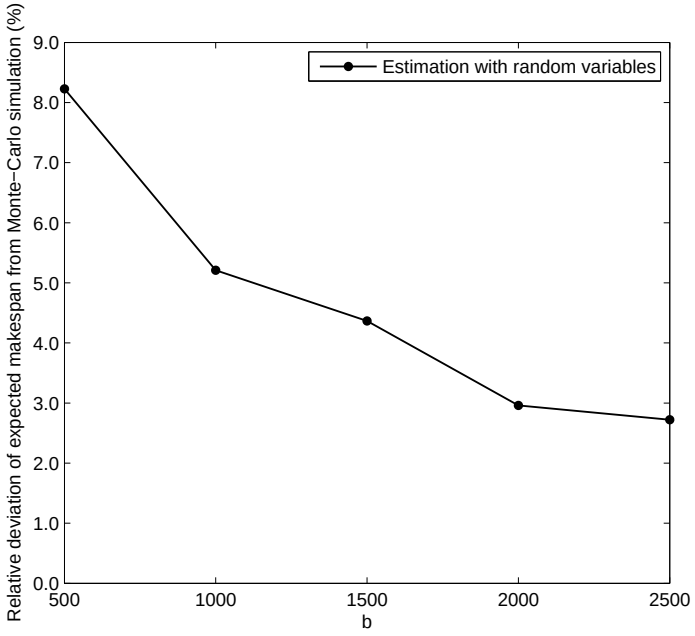


Figure 5.4: Relative deviation of expected makespan from Monte-Carlo simulation using different bin numbers

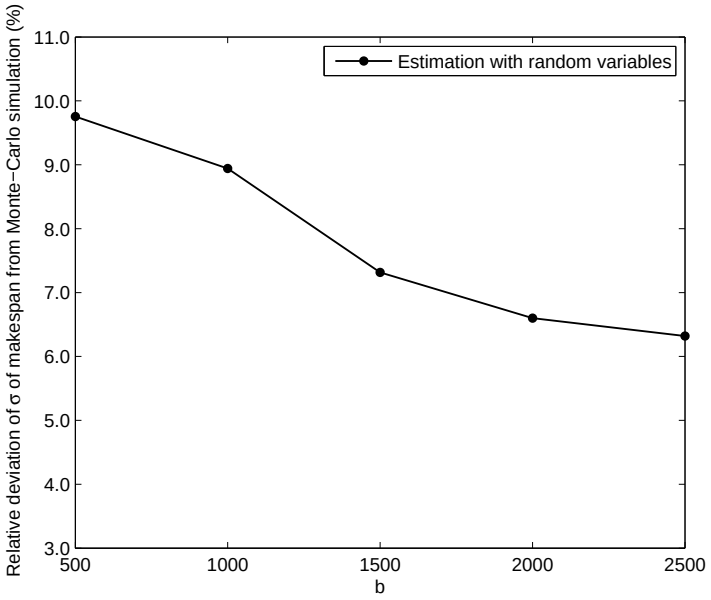


Figure 5.5: Relative deviation of σ of makespan from Monte-Carlo simulation using different bin numbers

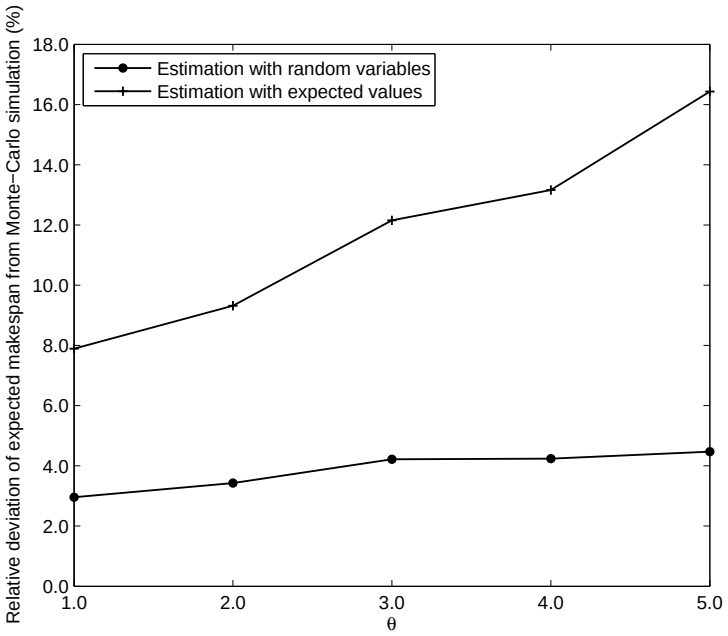


Figure 5.6: Relative deviation of expected makespan from Monte-Carlo simulation for graphs with different θ

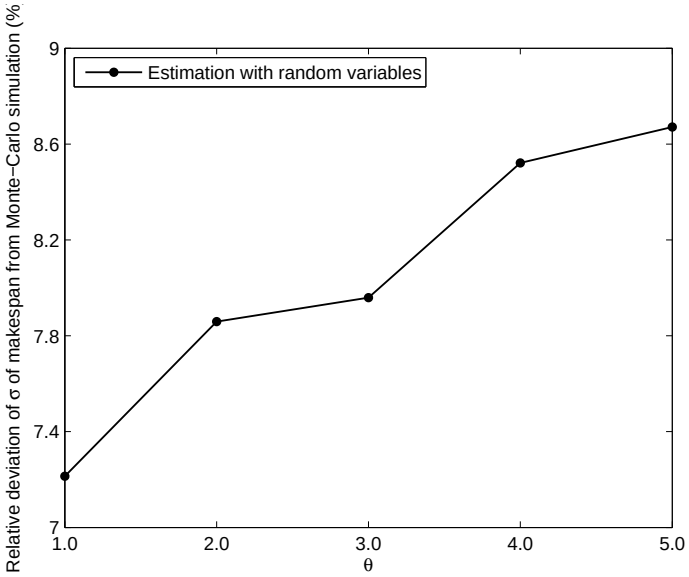


Figure 5.7: Relative deviation of σ of makespan from Monte-Carlo simulation for graphs with different θ

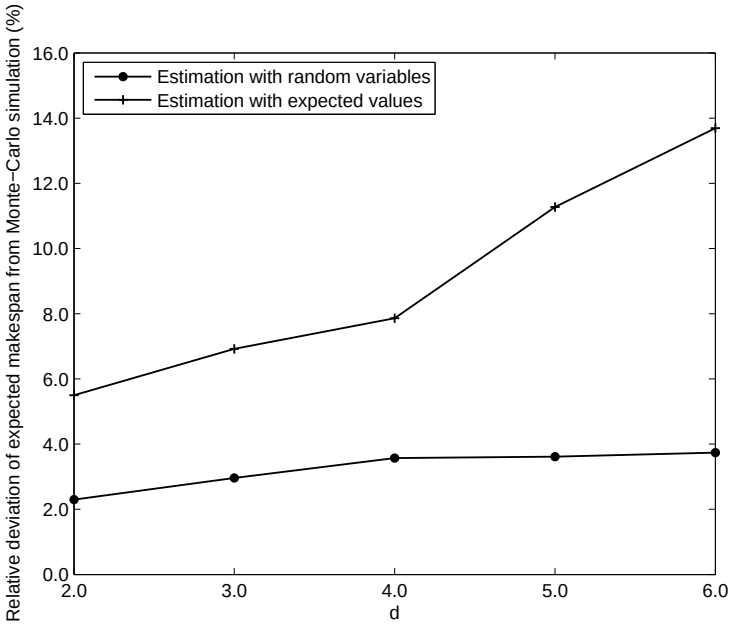


Figure 5.8: Relative deviation of expected makespan from Monte-Carlo simulation for graphs with different d

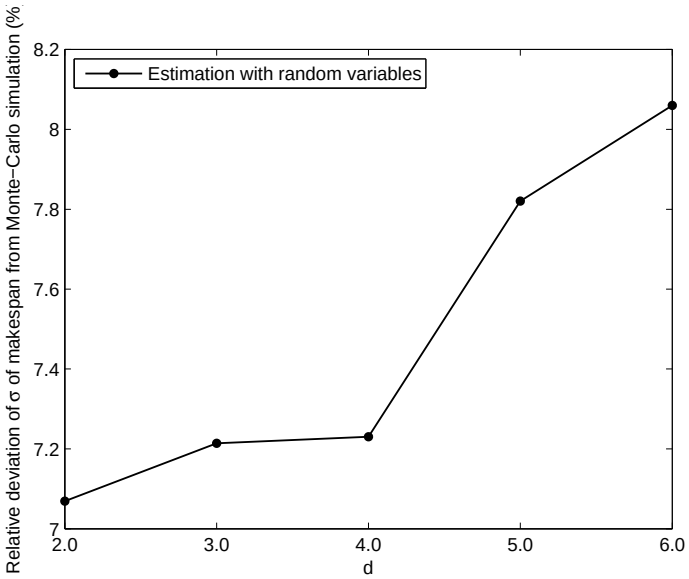


Figure 5.9: Relative deviation of σ of makespan from Monte-Carlo simulation for graphs with different d

distribution is an important factor in obtaining accurate estimation. Large bin numbers can better approximate the distribution of random variables. However, it will require a longer computation time to obtain the estimation. Figs. 5.4 and 5.5 shows relative deviation of the expectation and standard deviation of the makespan distribution using five values of bin number ($n = 100, \theta = 1.0, d = 3$). It is clear that using large bin number value can increase the accuracy of the estimation. However, as can be seen from the two figures, when the bin number is increased from 2000 to 2500, the relative deviation for both the expectation and standard deviation does not decrease substantially. This indicates that the error resulting from the discretization has been reduced significantly when the bin number is large than 2000.

The average scale parameter of the gamma distributions (θ) used to model the task execution time and data transfer rate can also affect the accuracy of the estimation. Figs. 5.6 and 5.7 depict how θ can affect the relative deviations. We observe that a large θ will increase the difficulty to obtain accurate estimation ($n = 100, b = 2000, d = 3$). During the discretization step, we fix the bin number to 2000 and the support to $[\max(0, \mu - 3\sigma), \mu + 3\sigma]$. Large θ implies that an arbitrary distribution will have large σ. For a fixed number of bins, a histogram can more accurately be approximated when the support is small, i.e., σ is small. As shown from Fig. 5.6, the relative deviation is increased from 3.05% to 4.22% when θ is changed from 1.0 to 5.0, which is not significant. This indicates that our choice of bin number (i.e., 2000) can control the error due to discretization appropriately. However, when we compute the makespan with only the

expected values of all the random variables, the relative deviation increases dramatically from 7.96% to 16.3%, which signifies the inadquacy of using such a method for estimating the makespan.

In real task graphs, the KIA is usually violated to a certain degree. In [67], the authors identified that the existence of isolated fork-join structures can cause the KIA to be violated. A fork-join structure (FJ) is a common way to achieve parallel processing. In a task graph, an FJ occurs when a single task node n_f *fork*s by sending data to multiple nodes that execute concurrently. As each branch completes its task, it waits on all the others to finish. The synchronization point is called a join (n_j). Let T represent the set of tasks in the FJ structure, excluding n_f and n_j. For a given schedule, an FJ structure is called an *isolated fork-join structure* (IFJ) if the following holds:

$$\forall n \in T, pred(n) \subseteq T \cup \{n_j\} \tag{5.26}$$

where $pred(n)$ denotes the immediate predecessors of n. The IFJ structures violate the KIA because the data that originated from the fork node n_f to the join node n_j does not merge with other data from tasks outside that IFJ. With the increase of average task node in-degree d, the chances of having multiple IFJs in a task graph are enhanced. From Figs. 5.8 and 5.9, we can conclude that a large d can unfavorably impact the accuracy of the estimation with respect to both the expectation and standard deviation ($n = 100, b = 2000, \theta = 1.0$). The relative deviation of expectation is increased from 2.14% to 3.87% when d rises from 2 to 6. For the standard deviation, the change is

from 7.05% to 8.04%. Compared with the makespan computed with expected values, the estimation still provides a better approximation for the makespan distribution.

5.5.2 SDS vs. DDS

In this section, we show the performance comparison between SDS and DDS. Note that in SDS, we use $E[T_S(c)]$ to assign the fitness of a chromosome c, while in DDS, $T_D(c)$ is used.

The simulation results are shown in Figs. 5.10 through 5.15. Figs. 5.10–5.12 show the effect of graph size on the makespan and robustness. It can be observed that SDS clearly outperforms HEFT and DDS with respect to reducing the makespan and increasing the robustness. It is evident that the improvement is further enhanced with the increase of the task graph size. For example, when the graph size is 20, the makespan obtained with SDS is reduced by 3.0% over DDS. At size 100, it is reduced by 11.9%. A similar effect can be perceived in terms of robustness. The performance improvement is due to the fact that DDS takes into account the variations in the task execution time and data transfer rate when making scheduling decisions. With the increase of task graph size, this variation becomes more important in order to obtain schedules with good performance.

Figs. 5.13 –5.15 show the effect of average scale parameter θ. Remember that the larger θ, the higher the variability of the system. It can be seen that with the increase of the system variability, a better performance gain of SDS over DDS and HEFT is achieved. Note that HEFT and DDS uses the expected task execution times and data

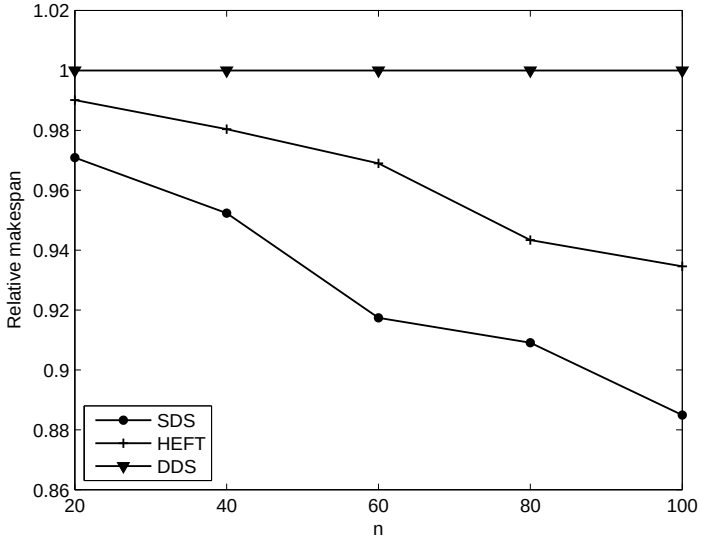


Figure 5.10: Relative makespan of schedules obtained with SDS, DDS, HEFT for graphs with different sizes

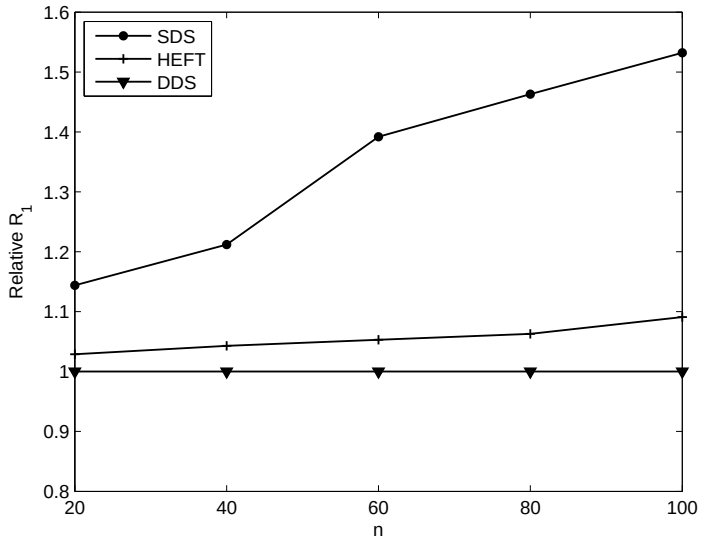


Figure 5.11: Relative R_1 of schedules obtained with SDS, DDS, HEFT for graphs with different sizes

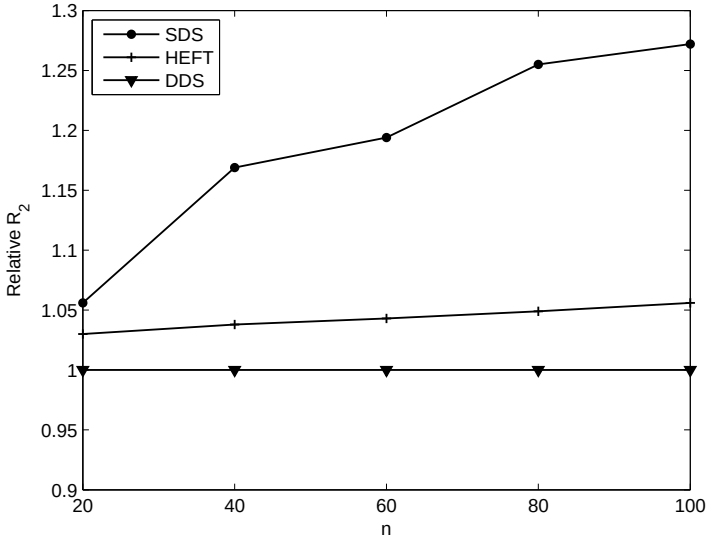


Figure 5.12: Relative R_2 of schedules obtained with SDS, DDS, HEFT for graphs with different sizes

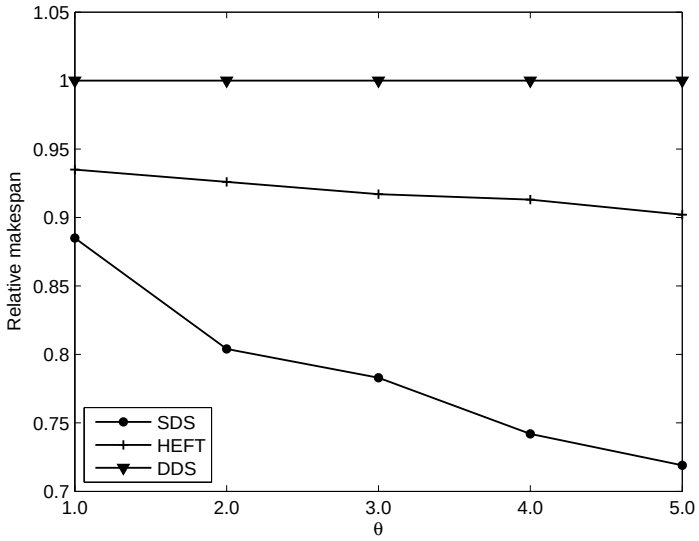


Figure 5.13: Relative makespan of schedules obtained with SDS, DDS, HEFT for graphs with different θ

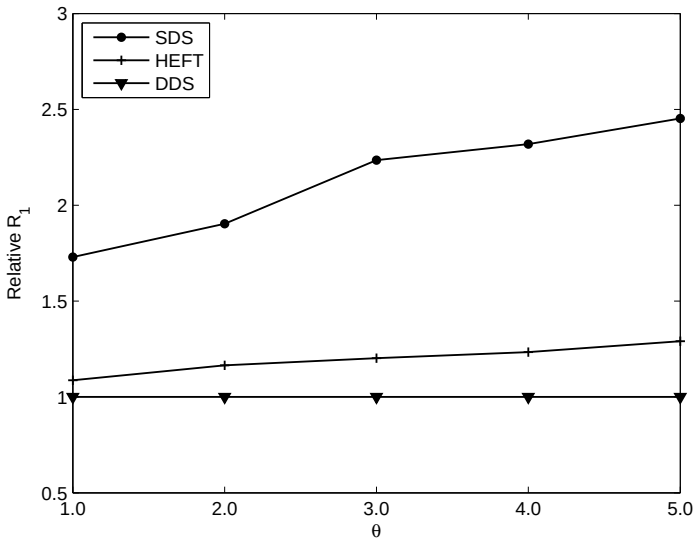


Figure 5.14: Relative R_1 of schedules obtained with SDS, DDS, HEFT for graphs with different θ

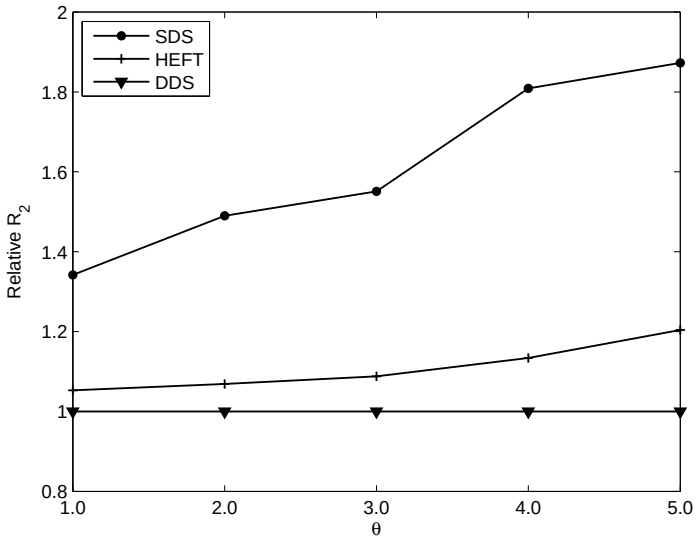


Figure 5.15: Relative R_2 of schedules obtained with SDS, DDS, HEFT for graphs with different θ

transfer rates. With the increase of system variability, the expected values becomes less relevant to the actual values of task execution time and data transfer rate. HEFT and DDS consequently generate poor schedules. However, SDS inherently use this variability to make scheduling decisions, thus it performs well even if the system has a high variability.

5.5.3 Bi-objective optimization

In this section, we investigate the bi-objective optimization problem, where both makespan and robustness need to be considered. Previously, we use only the $E[T_S(c)]$ as the fitness value of c in SDS. In the bi-objective optimization problem, the fitness value is determined using Eq. 5.6. Figs. 5.16 through 5.18 show the result of optimizing the makespan and R_1 (R_2) with different w. Here the first objective is the makespan and the second objective is R_1 (R_2). The Y-axis represents the logarithm value of the makespan(R_1, R_2) relative to the value at $w = 0, n = 20$ ($w = 0, n = 100$) in Fig. 5.16 (5.17, 5.18). The figures demonstrate that there is a trade off between reducing the makespan and increasing the robustness of the schedule. With the increase of w, more emphasis is placed on increasing the robustness. As can be seen from the figures, the makespan is smallest when $w = 0$ and largest when $w = 1.0$. In addition, when w increases, the makespan also increases. Moreover, the robustness R_1 is lowest when $w = 0$ and highest when $w = 1.0$. A similar trend can be observed from Fig. 5.18 when we use R_2 as the second objective.

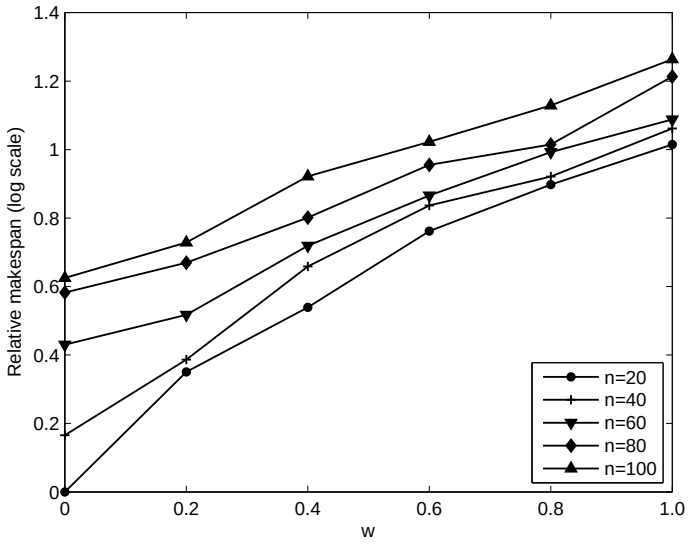


Figure 5.16: Relative makespan of schedules obtained with SDS for graphs with different sizes when w is a control parameter

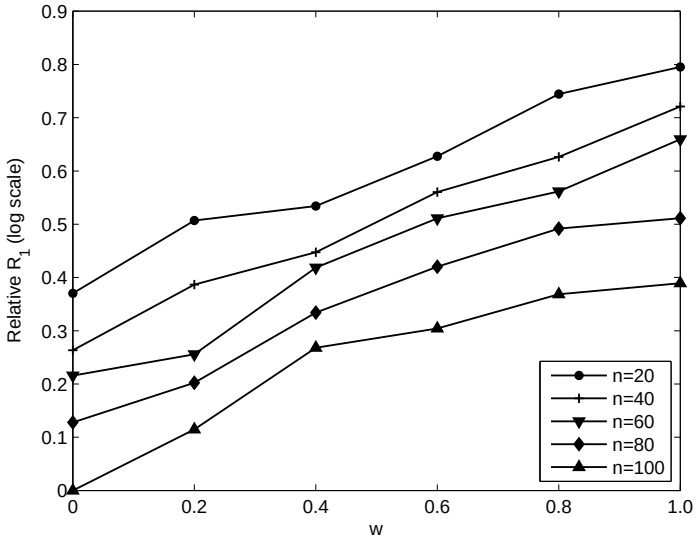


Figure 5.17: Relative R_1 of schedules obtained with SDS for graphs with different sizes when w is a control parameter

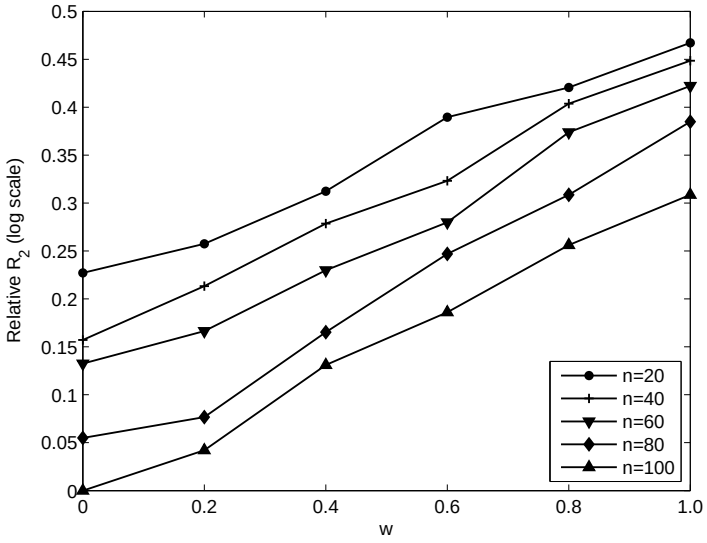


Figure 5.18: Relative R_2 of schedules obtained with SDS for graphs with different sizes when w is a control parameter

5.6 Summary

In this chapter, we investigated the problem of scheduling a DAG based task graph in HDCS where the performance characteristics of the system, such as the task executing times and data transfer rate, are stochastic. Traditional scheduling algorithms use the expected values of those performance characteristics. We proposed a genetic algorithm based approach (SDS) that utilizes the stochastic information during the setting of an individual's fitness value. However, the computation of $T_S(c)$ poses a challenge. It was shown that obtaining the exact makespan distribution is extremely difficult and impractical. We used an estimation method based on the Kleinrock Independence Approximation theorem to compute $T_S(c)$ numerically. Experiment results illustrated that

this approach is effective in obtaining makespan distributions with good accuracy.

We further compared the performance of SDS with DDS and HEFT. Because SDS takes into account the stochastic nature of task execution times and data transfer rate, it improves both the makespan and robustness of the obtained schedule significantly. Due to the accurate estimation of the makespan distribution in SDS, the algorithm makes better decisions in scheduling the task nodes. In addition, the makespan of the produced schedule is close to the makespan obtained when the schedule is carried out in the real dynamic environment. Thus, the robustness is also improved.

Because it is usually not possible to satisfy both goals of minimizing the makespan and maximizing the robustness of the schedule, we furthermore investigated the effectiveness of adjusting the weight w to control the trade-off between the two objectives. The bi-objective optimization algorithm facilitates the need for generating schedules that satisfy both objectives to some degree.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this dissertation, we developed several heuristics to schedule DAG-type applications in heterogeneous distributed computing systems. We first considered the case of scheduling DAGs onto HDCS where the processors have different capabilities. A list scheduling based heuristic is designed to consider the effect of resource scarcity. It has two distinct features. First, the task node weight assignment scheme takes into account the effect of Percentage of Capable Processors (PCP). For two task nodes with the same average computation cost, our weight assignment policy tends to give higher weight to the task with a smaller PCP. Secondly, during the processor selection phase, the algorithm adjusts the Earliest Finish Time (EFT) value by including the average communication cost between the current scheduling node and its children. Simulation results show that the algorithm has better performance compared with other heuristics.

We also investigated the problem of scheduling DAGs to both minimize the makespan and maximize the robustness. Due to the conflict of the two objectives, it is usually impossible to achieve both goals at the same time. We gave two definitions of robustness of a schedule based on tardiness and miss rate. We proved that slack is an effective metric to be used to adjust the robustness. The ϵ-constraint method was employed to solve the bi-objective optimization problem where minimizing the makespan and maximizing the slack are the two objectives. The overall performance of a schedule considering both makespan and robustness is defined such that users have the flexibility to place an emphasis on either objective. Experimental results validated the effectiveness of the proposed algorithm.

Next, we addressed the problem of scheduling a DAG application in non-deterministic HDCS. Most existing algorithms do not take into account the stochastic nature of task execution times and data transfer rates. We proposed a genetic algorithm based heuristic that accounts for the uncertainty of task execution times and data transfer rates by modeling them as random variables. The stochastic scheduling problem has an objective of both minimizing the expected value of makespan distribution and maximizing the robustness. We showed that the exact computation of the makespan distribution given a schedule is both extremely difficult and impractical due to task dependencies. Kleinrock Independence Approximation (KIA) is used to simplify the estimation. A numerical procedure was then described to compute the makespan distribution. We observed a performance improvement over deterministic algorithms from the experimental

results.

6.2 Future work

This dissertation mainly focused on the static task scheduling problem. In static scheduling, knowledge about the characteristics of the application such as task execution time, communication cost, and task dependencies are assumed to be available before execution, and the schedule is generated off-line. Admittedly, these assumptions limit the generality of such scheduling strategies in real world distributed systems. In a shared environment, predicting the processing power and communication bandwidth available to a given application is extremely difficult if at all possible. This also makes designing efficient static scheduling algorithms rather challenging. Chances are the estimations assumed by the static scheduler may no longer be kept the same during execution, which causes the application to perform poorly. One of the directions to extend our work is to develop dynamic scheduling algorithms to overcome the limitations imposed by static scheduling. Dynamic schedulers use information available at run-time to make scheduling decisions. However, the decision making process must be fast enough in order not to impact the application execution. In other words, the scheduling overhead must be minimized.

Another possible extension to our work is to consider the problem of scheduling real-time applications. In a real-time application, each task must be guaranteed to meet its timing constraint (deadline). For static task scheduling, the most common

objective is to minimize the overall makespan. However, in real-time task scheduling, the primary goal has shifted to meeting the deadline, otherwise the consequences could be catastrophic, especially for hard real-time applications. Recently, extensive research has focused on this subject [18, 55, 77, 80, 81, 93].

As the number of processors in today's HDCS continues to grow, the probability of processor failures also increases. Hence scheduling with fault tolerance is an important issue. In the future, we would like to explore scheduling DAG applications in such systems where the pattern of faults cannot be predicted. The goal here is to minimize the delay incurred by the tasks.

Bibliography

Bibliography

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, 1974.

[2] I. Ahmad and Y.-K. Kwok. A new approach to scheduling parallel programs using task duplication. In *Proc. Int'l Conf Parallel Processing*, volume 2, pages 47–51, 1994.

[3] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Trans. Parallel Distrib. Syst*, 9(9):872–892, 1998.

[4] A. H. Alhusaini, C. S. Raghavendra, and V. K. Prasanna. Run-time adaptation for grid environments. In *Proceedings 15th International Parallel and Distributed Processing Symposium.*, pages 864–874, 2001.

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

[6] S. Ali, S. M. Sait, and M. S.T. Benten. GSA: Scheduling and allocation using genetic algorithm. In *Proceedings of the 1994 European Design Automation Conference*, pages 84–89, Toronto, Canada, September 1994.

[7] S. Ali, H. J. Siegel, and A. A. Maciejewski. The robustness of resource allocation in parallel and distributed computing systems. In *ISPDC/HeteroPar'04*, 2004.

[8] S. Ali, H. J. Siegel, M. Maheswaran, and D. A. Hensgen. Task execution time modeling for heterogeneous computing systems. In *Heterogeneous Computing Workshop*, pages 185–199, 2000.

[9] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[10] D. Berleant. Automatically verified reasoning with both intervals and probability density functions. *Interval Computations*, (2):48–70, 1993.

[11] D. Berleant and H. Cheng. A software tool for automatically verified operations on intervals and probability distributions. *Reliable Computing*, 4(1):71–82, 1998.

[12] D. Berleant and C. Goodman-Strauss. Bounding the results of arithmetic operations on random variables of unknown dependency using intervals. *Reliable Computing*, 4(2):147–165, 1998.

[13] D. Berleant, L. Xie, and J. Zhang. Statool: A tool for distribution envelope determination (DEnv), an interval-based algorithm for arithmetic on random variables. *Reliable Computing*, 9(2):91–108, 2003.

[14] F. Berman, H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, W. Deng, J. Dongarra, L. Johnsson, K. Kennedy, C. Koelbel, B. Liu, X. Liu, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, C. Mendes, A. Olugbile, M. Patel, D. Reed, Z. Shi, O. Sievert, H. Xia, and A. YarKhan. New grid scheduling and rescheduling methods in the GrADS project. *International Journal of Parallel Programming (IJPP)*, 33(2-3):209–229, 2005.

[15] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. In *CCGRID*, pages 759–767, 2005.

[16] L. Bölöni and D. Marinescu. Robust scheduling of metaprograms. *Journal of Scheduling*, 5(5), 2002.

[17] W. F. Boyer and G. S. Hura. Dynamic scheduling in distributed heterogeneous systems with dependent tasks and imprecise execution time estimates. In *Proceedings of 16th IASTED International Conference, Parallel and distributed computing and systems*, Cambridge, MA, 11 2004.

[18] T. Braun, H. Siegel, and A. Maciejewski. Static mapping heuristics for tasks with dependencies, priorities, deadlines, and multiple versions in heterogeneous envi-

ronments. In *16th International Parallel and Distributed Processing Symposium (IPDPS '02)*, pages 78–78. IEEE, April 2002.

[19] T. D. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, and B. Yao. A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pages 330–335, West Lafayette, IN, 1998.

[20] T. D. Braun, H. J. Siegel, N. Beck, L. L. Boloni, A. I. Reuther, M. D. Theys, B. Yao, R. F. Freund, M. Maheswaran, J. P. Robertson, and D. Hensgen. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *HCW '99: Proceedings of the Eighth Heterogeneous Computing Workshop*, page 15, Washington, DC, USA, 1999. IEEE Computer Society.

[21] P. Brucker. *Scheduling Algorithms*. SpringerVerlag, 2004.

[22] H. Casanova. Network modeling issues for grid application scheduling. *Int. J. Found. Comput. Sci*, 16(2):145–162, 2005.

[23] V. Chankong and Y. Haimes. *Multiobjective Decision Making Theory and Methodology*. Elsevier Science, Newe York, 1983.

[24] A. G. Colombo and R. J. Jaarsma. A powerful numerical method to combine random variables. *IEEE TRansactions on Reliability*, 2(R-29):126–129, 1980.

[25] R.C. Corrêa, A. Ferreira, and P. Rebreyend. Scheduling multiprocessor tasks with genetic algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):825–837, 1999.

[26] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. Addison Wesley, Essex, England, 2005.

[27] R. L. Daniels and P. Kouvelis. Robust scheduling to hedge against processing time uncertainty in single-stage production. *Management Science*, 41(2):363–376, 1995.

[28] S. Darbha and D. P. Agrawal. A task duplication based scalable scheduling algorithm for distributed memory systems. *J. Parallel Distrib. Comput*, 46(1):15–27, 1997.

[29] S. Darbha and S. Pande. A robust compile time method for scheduling task parallelism on distributed memory machines. *The Journal of Supercomputing*, 12(4):325–347, 1998.

[30] A. J. Davenport and J. C. Beck. A survey of techniques for scheduling with uncertainty. *Preprint*, 2000.

[31] L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, NY, 1991.

[32] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, Chichester, UK, 2001.

[33] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In Marios D. Dikaiakos, editor, *Grid Computing, Second European Across Grids Conference, AxGrids 2004, Nicosia, Cyprus, January 28-30, 2004, Revised Papers*, volume 3165 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2004.

[34] A. Doğan and F. Özgüner. Genetic algorithm based scheduling of meta-tasks with stochastic execution times in heterogeneous computing systemst1. *Cluster Computing*, 7(2):177–190, 2004.

[35] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Distrib. Comput.*, 9(2):138–153, 1990.

[36] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, Englewood Cliffs, 1994.

[37] D. England, J. Weissman, and J. Sadagopan. A new metric for robustness with application to job scheduling. *HPDC-14*, pages 135–143, 2005.

[38] M. M. Eshaghian, editor. *Heterogeneous Computing*. Artech House Publishers, Boston, 1996.

[39] S. Ferson. What monte carlo methods cannot do. *Human and Ecological Risk Assessment*, 2(4):990–1007, 1996.

[40] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *HPDC*, pages 365–376, 1997.

[41] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[42] R. F. Freund and H. J. Siegel. Introduction: Heterogeneous processing – guest editors' introduction. *IEEE Computer*, 26(6):13–17, June 1993.

[43] S. Fujita and H. Zhou. Multiprocessor scheduling problem with probabilistic execution costs. In *ISPAN: Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*. IEEE Computer Society Press, 2000.

[44] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[45] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, November 1984.

[46] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, December 1992.

[47] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, Reading, Massachusetts, 1989.

[48] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93, San Mateo, 1991. Morgan Kaufmann.

[49] J. H. Holland. *Adaption in Natural and Artificial Systems.* MIT Press, 1992.

[50] E. S. H. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 5(2):113–120, 1994.

[51] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, 1989.

[52] G. E. Ingram, E. L. Welker, and C. R. Herrmann. Designing for reliability based on probabilistic modeling using remote access computer systems. In *Proc. 7th Reliabiltiy and Maintainability Conference*, pages 492–500. American Society of Mechanical Engineers, 1968.

[53] H. A. James. *Scheduling in Metacomputing Systems.* PhD thesis, University of Adelaide, Adelaide,Australia, 1999.

[54] S. A. Jarvis, L. He, D. P. Spooner, and G. R. Nudd. The impact of predictive inaccuracies on execution scheduling. *Performance Evaluation*, 60(1-4):127–139, 2005.

[55] V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser. Dynamic scheduling for soft real-time distributed object systems. In *ISORC*, pages 114–121. IEEE Computer Society, 2000.

[56] A. Kamthe and S.-Y. Lee. A stochastic approach to estimating earliest start times of nodes for scheduling DAGs on heterogeneous distributed computing systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 2005.

[57] S. Kaplan. On the method of discrete probability distributions in risk and reliability calculations, applications to seismic risk assessment. *Risk Analysis*, 1(3):189–196, 1981.

[58] T. Kidd and D. Hensgen. Why the mean is inadequate for accurate scheduling decisions. In *Parallel Architectures, Algorithms, and Networks, 1999. (I-SPAN '99) Proceedings. Fourth InternationalSymposium on*, pages 262–267, Perth/Fremantle, WA, 1999.

[59] S. J. Kim and J. C. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. In *Proceedings of International Conf. Parallel Processing*, volume 2, pages 1–8, 1988.

[60] L. Kleinrock. *Communication Nets: Stochastic Message Flow and Delay*. McGraw-Hill, 1964.

[61] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *IEEE Softw.*, 5(1):23–32, 1988.

[62] Y.-K. Kwok and I. Ahmad. Exploiting duplication to minimize the execution times of parallel programs on message-passing systems. In *Proceedings of the 6th Symposium on Parallel and Distributed Processing*, pages 426–433, Los Alamitos, CA, USA, October 1994. IEEE Computer Society Press.

[63] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7(5):506–521, 1996.

[64] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.*, 59(3):381–422, 1999.

[65] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[66] V. J. Leon, S. D. Wu, and R. H. Storer. Robustness measures and robust scheduling for job shops. *IIE Transactions*, 26(5):32–43, 1994.

[67] Y. A. Li and J. K. Antonio. Estimating the execution time distribution for a task graph in a heterogeneous computing system. In *Heterogeneous Computing Workshop*, pages 172–184. IEEE Computer Society, 1997.

[68] J. Liou and M. A. Falls. An efficient clustering heuristic for scheduling dags on multiprocessors. In *Proc. Symp. Parallel and Distributed Processing*, 1996.

[69] G. Q. Liu, K. L. Poh, and M. Xie. Iterative list scheduling for heterogeneous computing. *J. Parallel Distrib. Comput.*, 65(5):654–665, 2005.

[70] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Heterogeneous Computing Workshop*, pages 57–69, 1998.

[71] R. E. Moore. *Interval Analysis*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1966.

[72] R. E. Moore. Risk analysis without monte carlo methods. *Freiburger Intervall-Berichte*, 84(1):1–48, 1984.

[73] R. Moreno and A. B. Alonso-Conde. Job scheduling and resource management techniques in economic grid environments. In F. Fernández Rivera, Marian Bubak, A. Gómez Tato, and Ramon Doallo, editors, *European Across Grids Conference*, volume 2970 of *Lecture Notes in Computer Science*, pages 25–32. Springer, 2003.

[74] A. Moukrim, E. Sanlaville, and F. Guinand. scheduling with communication delays and on-line disturbances. In *Euro-Par'99, LNCS 1685*, pages 350–357, 1999.

[75] A. Moukrim, E. Sanlaville, and F. Guinand. parallel machine scheduling uncertain communication delays. *RAIRO Operations Research*, 37:1–16, 2003.

[76] R. B. Nelson. An introduction to copulas. In *Lecture notes in statistics 139*. Berlin:Springer, 1999.

[77] M. A. Palis. On the competitiveness of online real-time scheduling with rate of progress guarantees. *International Journal of Foundations of Computer Science*, 14(3):359–370, 2003.

[78] G.-L. Park, B. Shirazi, and J. Marquis. DFRN: A new approach for duplication based scheduling for distributed memory multiprocessor systems. In *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*, pages 157–166, Washington, DC, USA, 1997. IEEE Computer Society.

[79] N. Policella. *Scheduling with uncertainty: A proactive approach using Partial Order Schedules*. PhD thesis, University of Rome "La Sapienza", Rome, March 2005.

[80] X. Qin and H. Jiang. Dynamic, reliability-driven scheduling of parallel real-time jobs in heterogeneous systems. In *2001 International Conference on Parallel Processing (ICPP '01)*, pages 113–122. IEEE, September 2001.

[81] X. Qin and H. Jiang. A dynamic and reliability-driven scheduling algorithm for parallel real-time jobs executing on heterogeneous clusters. *J. Parallel Distrib. Comput*, 65(8):885–900, 2005.

[82] A. Radulescu and A. J. C. Van Gemund. Fast and effective task scheduling in heterogeneous systems. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 229, Washington, DC, USA, 2000. IEEE Computer Society.

[83] S. Ranaweera and D. Agrawal. A scalable task duplication based scheduling algorithm for heterogeneous systems. In *Proceedings of 2000 International Conference on Parallel Processing (29th ICPP'00)*, Toronto, Canada, August 2000. Ohio State Univ.

[84] S. Ranaweera and D. P. Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In *14th International Parallel and Distributed Processing Symposium (SPDP'2000)*, pages 445–450, Washington - Brussels - Tokyo, May 2000. IEEE.

[85] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Proceedings of 13th heterogeneous computing workshop (HCW2004)*, Santa Fe, NM, April 2004.

[86] R. Sakellariou and H. Zhao. A low-cost rescheduling policy for efficient mapping of workflows on grid systems. *Scientific Programming*, 12(4):253–262, 2004. U. Manchester.

[87] N. Sample, P. Keyani, and G. Wiederhold. Scheduling under uncertainty: Planning for the ubiquitous grid. In Farhad Arbab and Carolyn L. Talcott, editors, *Coordination Models and Languages, 5th International Conference, COORDINATION 2002, YORK, UK, April 8-11, 2002, Proceedings*, volume 2315 of *Lecture Notes in Computer Science*, pages 300–316. Springer, 2002.

[88] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, Massachusetts, 1989.

[89] J. M. Schopf and F. Berman. Stochastic scheduling. In *Proceedings of Supercomputing'99 (CD-ROM)*, Portland, OR, November 1999. ACM SIGARCH and IEEE.

[90] J. M. Schopf and F. Berman. Using stochastic information to predict application behavior on contended resources. *International Journal of Foundations of Computer Science*, 12(3):341–363, 2001.

[91] S. P. Shah, D. YM He, J. N. Sawkins, J. C. Druce, G. Quon, D. Lett, G. XY Zheng, T. Xu, and BF F. Ouellette. Pegasys : software for executing and integrating analyses of biological sequences. *BMC Bioinformatics*, 5(40), 2004.

[92] Zhiao Shi and Jack Dongarra. Scheduling workflow applications on processors with different capabilities. *Future generation computer systems*, 22(6):665–675, 2006.

[93] B. Shirazi, H. Y. Youn, and D. M. Lorts. Evaluation of static scheduling heuristics for real-time multiprocessing. *Parallel Processing Letters*, 5(4):599–610, December 1995.

[94] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):175–187, 1993.

[95] H. Singh and A. Youssef. Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *Proc. Heterogeneous Computing Workshop*, pages 86–97, 1996.

[96] M. D. Springer. *The Algebra of Random Variables*. Wiley series in Probability and Mathematical Statistics. John Wiley & Sons, New York, NY, USA, 1979.

[97] Condor team. The direct acyclic graph manager (DAGMan), 2002. http://www.cs.wisc.edu/condor/dagman/.

[98] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002.

[99] P. J. M. van Laarhoven and E. H. L. Aarts. *Simulated annealing: theory and applications*. Kluwer, Dordrecht, 1992.

[100] L. Wang, H. J. Siegel, and V. P. Roychowdhury. A genetic-algorithm-based approach for task matching and scheduling in heterogeneous computing environments. In *Proc. Heterogeneous Computing Workshop*, 1996.

[101] 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.

[102] R. C. Williamson. Interval arithmetic and probabilistic arithmetic. In C. Ullrich, editor, *Contrib. Comp. Arith. Self-Val.*, pages 67–80, 1990.

[103] R. C. Williamson and T. Downs. Probabilistic arithmetic. I. numerical methods for calculating convolutions and dependency bounds. *Int. J. Approx. Reasoning*, 4(2):89–158, 1990.

[104] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1(1):119–132, 1998.

[105] A. S. Wu, H. Yu, S. Jin, K.-C. Lin, and G. Schiavone. An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):824–834, 2004.

[106] M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(3):330–343, 1990.

[107] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel Distrib. Syst.*, 5(9):951–967, 1994.

[108] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *J. of Grid Computing*, (3):171–200, 2006.

[109] H. Zhao and R. Sakellariou. An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In *Proceedings of 9th International Euro-Par Conference*. Springer-Verlag, 2003.

[110] A. Y. Zomaya, C. Ward, and B. Macey. Genetic scheduling for parallel processor systems: comparative studies and performance issues. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):795–812, 1999.

Vita

Zhiao Shi was born in Ningbo, China. He received a Bachelor of Engineering degree from Beijing University of Chemical Technology in 1996 and a Master of Science degree from Kansas State University in 2000, both in Chemical Engineering.

He moved to the University of Tennessee to pursue a doctoral degree in January 2001. He worked in the Innovative Computing Laboratory (ICL) as a Graduate Research Assistant under the guidance of Dr. Jack Dongarra. His current research interests include parallel and distributed computing, grid computing, task scheduling in heterogeneous environments. Zhiao Shi is expected to receive a Doctor of Philosophy degree in Computer Science in December 2006.