4-8-2005

A Game Theoretic Framework for Dynamic Task Scheduling in Distributed Heterogeneous Computing Systems

Vasanth Kumar Ramesh
University of South Florida

A GAME THEORETIC FRAMEWORK FOR DYNAMIC TASK SCHEDULING IN DISTRIBUTED HETEROGENEOUS COMPUTING SYSTEMS

by

VASANTH KUMAR RAMESH

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: N. Ranganathan Ph.D.
Srinivas Katkoori Ph.D.
Soontae Kim Ph.D.

Date of Approval:
April 8, 2005

Keywords: Variable Structure Stochastic Automation, Game Theory, Nash Equilibrium, Auctions, Task Assignment

DEDICATION

To my beloved parents – *Sri. M. S. Ramesh* and *Smt. Chandra Ramesh*.

ACKNOWLEDGEMENTS

Words cannot express my emotions and it was almost impossible for me to express my heartfelt gratitude to Dr. N. Ranganathan. It was just the motivation I needed to break my dry routine. I am truly blessed to have him as a mentor and friend. My mentor has led me to dedicate my efforts to the pursuit of research and his contribution to my work is something that cannot be gauged in any scale. This is the result of many hours of discussions and innumerable telephone calls. His enthusiastic approach to life and studies has given me a new outlook on a research career.

I would like to thank Dr. Srinivas Katkoori and Dr. Soontae Kim for their critical remarks and suggestions to improve the work.

I am very grateful to Prof. Waran who has always been motivating me through his golden words of advice. He has always boosted my confidance during hard times.

I express my heartfelt gratitude to my mother – my first Guru, for her love, affection and constant encouragement and care throughout. I would like to thank my younger brother Prem ,for his love and concern.

Finally I would like to thank Pradeep, Mali and Naren for their timely help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

A GAME THEORETIC FRAMEWORK FOR DYNAMIC TASK SCHEDULING IN DISTRIBUTED HETEROGENEOUS COMPUTING SYSTEMS

Vasanth Kumar Ramesh

ABSTRACT

Heterogeneous Computing (HC) systems achieve high performance by networking together computing resources of diverse nature. The issues of task assignment and scheduling are critical in the design and performance of such systems. In this thesis, an auction based game theoretic framework is developed for dynamic task scheduling in HC systems. Based on the proposed game theoretic model, a new dynamic scheduling algorithm is developed that uses auction based strategies. The dynamic scheduling algorithm yields schedules with shorter completion times than static schedulers while incurring higher scheduling overhead. Thus, a second scheduling algorithm is proposed which uses an initial schedule generated with a learning automaton based algorithm, and then heuristics are used to identify windows of tasks within the application that can be rescheduled dynamically during run time. The algorithm yields significantly better completion times compared to static scheduling while incurring lesser overhead than a purely dynamic scheduler. Several different heuristics are investigated and compared in terms of how they impact the overall scheduler performance. Experimental results indicate that the proposed algorithms perform significantly better than previous algorithms reported in the literature.

CHAPTER 1

INTRODUCTION

In this chapter, the field of heterogenous computing (HC) is introduced, the various classifications, advantages and disadvantages are discussed. Several issues related to the HC system are addressed and the motivation for task-assignment and scheduling in HC system is discussed. Also, an introduction to task assignment and scheduling, which is the focus of this thesis, is presented.

1.1 Introduction

Scientists and engineers have always been striving hard to build machines for high performance. Technology growth has led to the development of processors with hundreds of millions of devices within a die. Traditionally, computer architects have been focussed on developing homogenous computing models to deliver superior performance. Hence their goal was to develop a single architecture that could satisfy the requirements of a wide range of applications. Modern day applications have almost saturated the capabilities of such architectures. Currently we have come to believe that single architecture computers are no longer suitable for high performance. Studies have shown that most of the time, the processor executes code for which it is poorly suited[1]. One way to overcome this problem is to build a system with several types of architectures and then attempt to match the requirement of application to the suitable processor. Hence, the focus is shifting towards heterogenous computing systems.

In [5] HC is defined as

> *“tuned use of diverse processing hardware to meet distinct computational needs.”*

Another popular definition for HC system is given in [6] as

> *“the well orchestrated and coordinated effective use of a suite of diverse high-performance machines to provide super-speed processing for computationally demanding tasks with diverse computing needs.”*

From these definitions it is clear that the key feature in a HC system is the *diversity* of the processor architectures. High performance is achieved by exploiting this key feature. Figure 1.1 illustrates the concept of heterogenous computing.

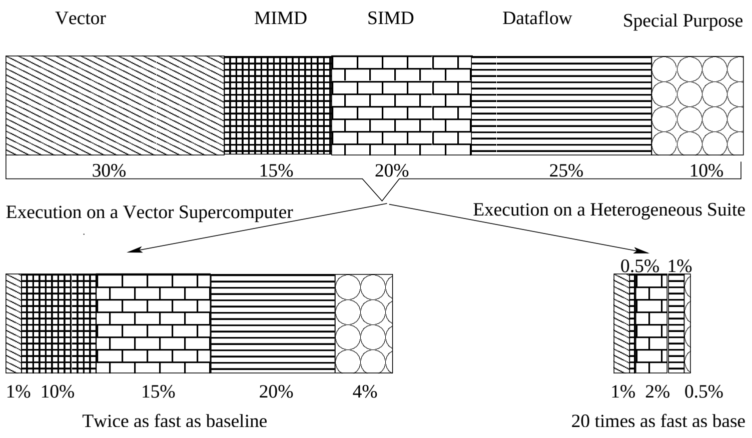


Figure 1.1. Task Profiling Example Illustrating the Advantage of HC Systems [1]

It is a known fact that achieving usable as opposed to peak performance is a grand challenge problem [7]. Many High-Performance Computing (HPC) systems achieve only a fraction of their peak performance. Another feature that attracts scientists to use HC systems for supercomputing is its economic viability. Instead of replacing systems with high cost supercomputers, HC offers a structured methodology to integrate existing systems for high performance computing. Distributed Computing and Network Computing have networks that vary in the topology and bandwidth and hence heterogeneity in these systems is partly justified. These systems are homogenous with respect to the computational facilities and lack the full spectrum of a HC system. The fundamental heterogenous processing heuristic is:

> ”First evaluate suitability of tasks to processor types, then load-balance among selected machines for the final assignment.”[5]

1.2 Taxonomy of Heterogenous Computing Systems

Heterogenous computing is broadly classified into two categories, namely:

1. System Heterogenous Computing
2. Network Heterogenous Computing

Figure 1.2 illustrates the classification of heterogenous computing systems. They are discussed briefly in the following sections.

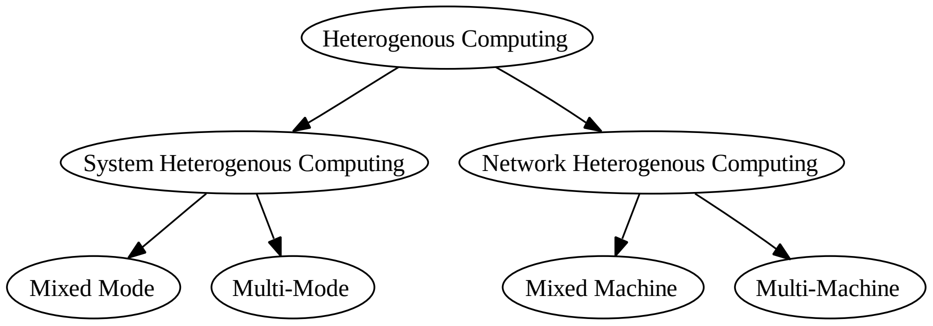


Figure 1.2. Classification of Heterogenous Computing Systems [2]

1.2.1 System Heterogenous Computing

System Heterogenous Computing (SHC) consists of a single supercomputer that can be configured to execute tasks in different modes such as SIMD and MIMD. Figures 1.3 and 1.4 illustrate a distributed memory SIMD and a distributed memory MIMD respectively. The SHC can be further classified into *mixed mode* and *multi mode* systems.

Mixed Mode: In mixed mode operation, the processing elements switch between different modes of operation. But, at any given time, the system can execute in only one of the modes. The PASM [8] system is an example of SHC system.

Multi-Mode: In multi-mode operation, the processing elements can execute in different modes at the same time. The Image Understanding Architecture [9] system is a multi-level system where each level comprises of processing elements configured in different modes. The tasks are mapped onto the different layers and they can co-exist.

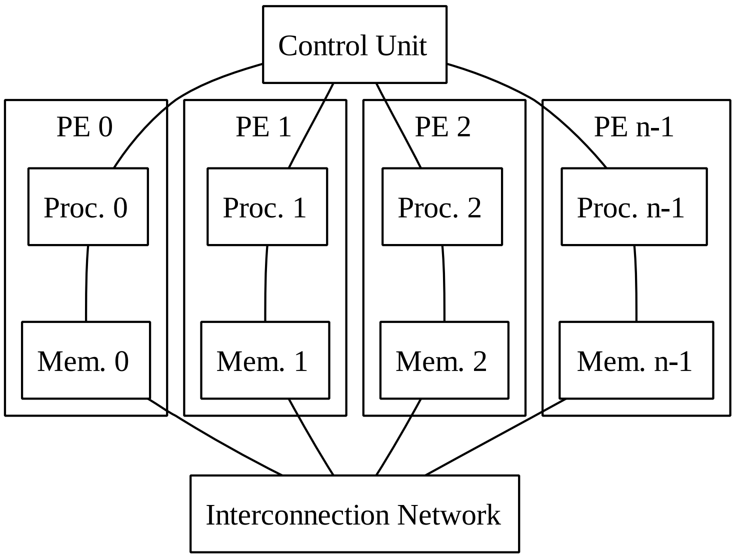


Figure 1.3. Structure of a Distributed Memory SIMD [2]

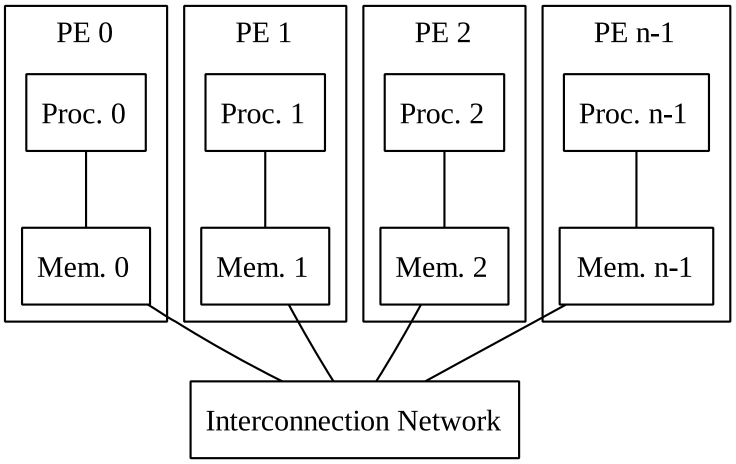


Figure 1.4. Structure of a Distributed Memory MIMD [2]

1.2.2 Network Heterogenous Computing

The Network Heterogenous Computing (NHC) system consists of autonomous computers that are connected in a network. These computers have the ability to execute tasks concurrently. NHC systems are further classified as *mixed machine* and *multi machine* systems. Multi machine NHC systems are studied as a special case of mixed machine NHC systems. There are three layers in NHC, namely: *Network layer*, *Communication Layer* and *Processing Layer*.

Network Layer: This is the lowest layer. This layer takes care of handling all the issues related to connecting two computers such as which routing path to take, which routers to use and which protocols to follow and other issues related to computer networks.

Communication Layer: This is the intermediate layer and provides mechanisms for computers to communicate with each other. This layer also provides utilities and tools that enable user to view a group of computer as a single virtual machine. Some examples of these tools include Parallel Virtual Machine *PVM*[10], Message Passing Interface *MPI* [11],Linda [12], p4[13], Mentat[14], HeNCE [15, 6] and Java programming language. These tools run as a system daemons and provide service to any request from the application process.

Processing Layer: This layer provides tools and techniques that ensure efficient execution of the applications. This layer controls the performance of the heterogenous system. Examples of some services provided in this layer include task decomposition, task mapping and load balancing.

1.3 Issues in Heterogenous Computing Systems

There are various issues to be addressed in a heterogenous computing system. These issues are enumerated below:

- Algorithm design

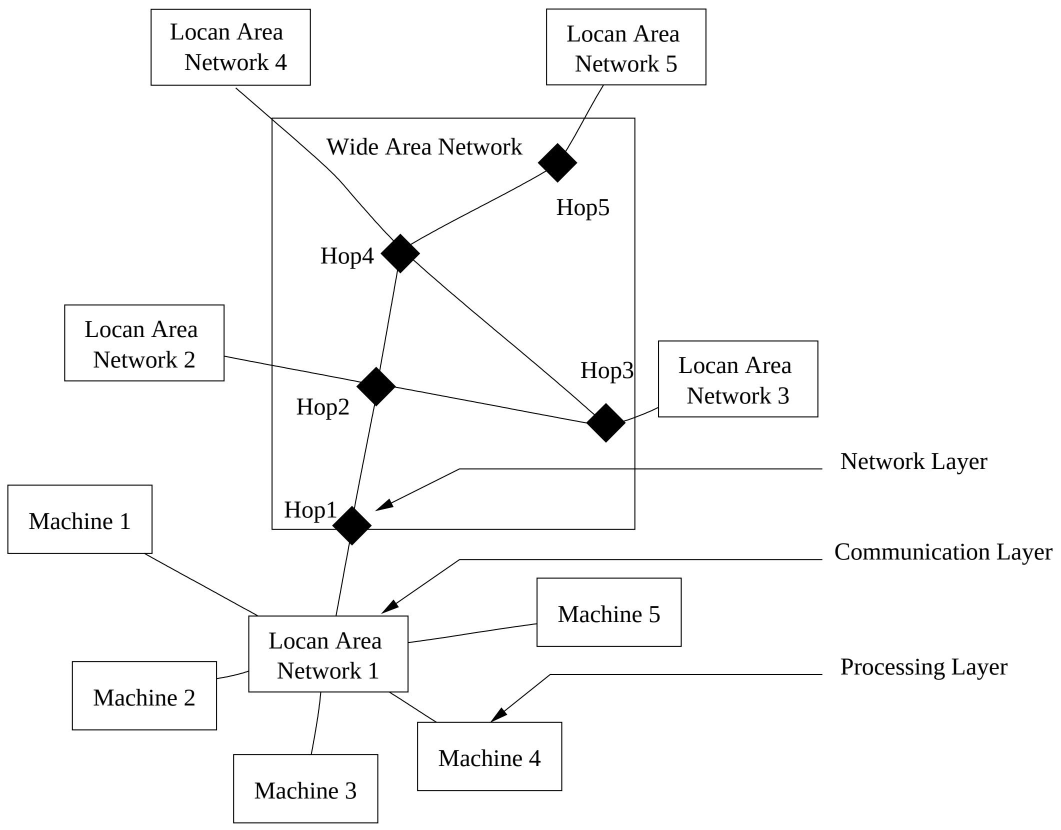


Figure 1.5. Structure of NHC System [2]

- Code-type profiling and analytical benchmarking
- Code Partitioning
- Task mapping
- Network requirement
- Programming environment
- Performance evaluation

A brief overview of these issues is presented next. A sequence of steps involved in [16] is depicted in figure 1.6.

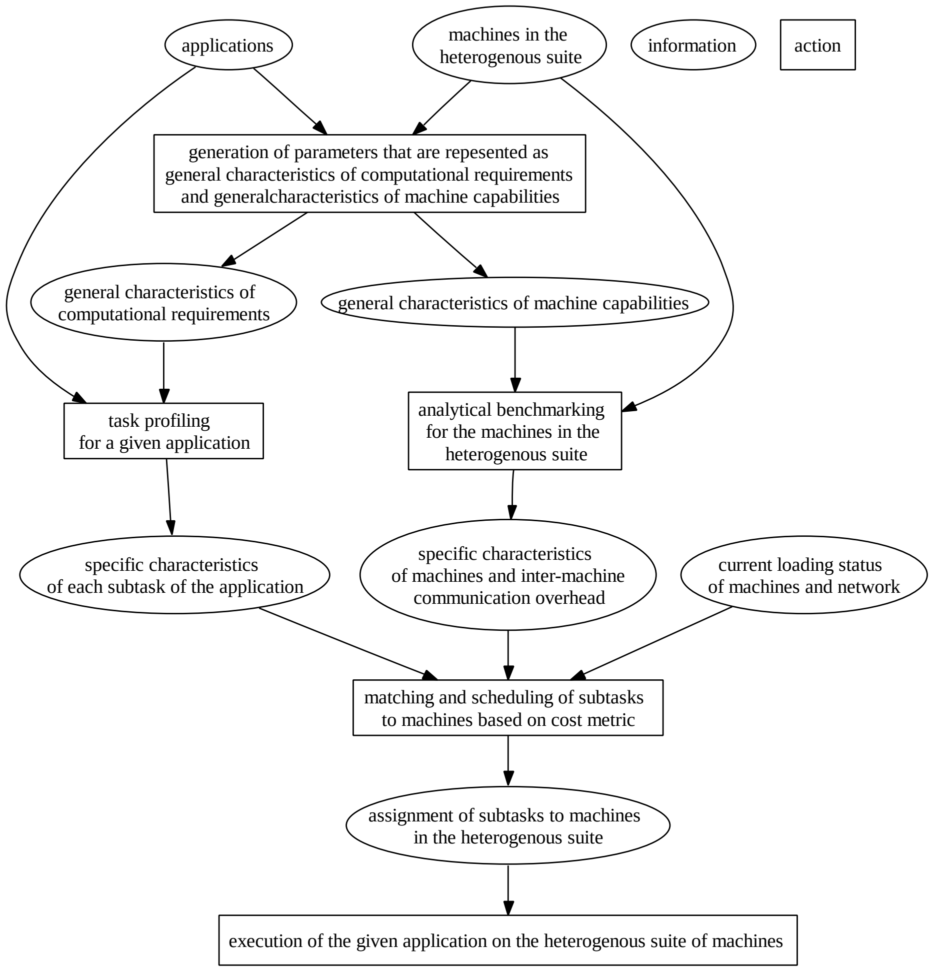

applications
machines in the heterogenous suite
information
action
generation of parameters that are repesented as general characteristics of computational requirements and generalcharacteristics of machine capabilities
general characteristics of computational requirements
general characteristics of machine capabilities
task profiling for a given application
analytical benchmarking for the machines in the heterogenous suite
specific characteristics of each subtask of the application
specific characteristics of machines and inter-machine communication overhead
current loading status of machines and network
matching and scheduling of subtasks to machines based on cost metric
assignment of subtasks to machines in the heterogenous suite
execution of the given application on the heterogenous suite of machines

Figure 1.6. Conceptual Model of the Assignment of Subtasks to Machines in a HC System Environment (adapted from [1])

1.3.1 Algorithm Design

Many parameters must be taken into account in order to design an efficient algorithm for heterogenous system. They are :

- The various alternatives to solve the problem must be explored. The application must be analyzed and its inherent heterogeneity must be exploited by the algorithm.
- The number of machines in the suite and their computational abilities, instruction set and architectural features must be taken into account.
- The cost of communicating through the network is an important factor that should not be overlooked when designing an algorithm for HC system.

1.3.2 Code-type Profiling and Analytical Benchmarking

Code type profiling or task profiling is defined as a method to quantify the types of computations that are present in an application [17].
Analytical benchmarking is defined as the procedure that provides a measure of how well the available machines in the HC suite perform on the given set of code-types [17].

1.3.3 Partition

Partitioning an application in a HC system involves solving two sub-problems. They are *Parallelism Detection* and *Clustering*. *Parallelism Detection* involves determination of the kind of parallelism that is present in the given application. *Clustering* combines several operations in the application into a single module.

1.3.4 Task Mapping

Task mapping involves the twin process of task assignment and task scheduling. Tasks assignment is also known as task matching. *Task assignment* determines which machine is best suited to execute a given task.

Task scheduling determines when to execute the task so that a performance cost metric of the system is optimized. The task mapping algorithms can be broadly classified into two categories: Static and dynamic. A detailed survey of these techniques is presented in chapter 2.

1.3.5 Network Requirement

The interconnection network impacts the performance of a HC system. The communication requirements of a HC system are of the order of gigabits per second, whereas modern day LANs can provide only few tens of megabits per second. Hence LANs are unsuitable for HC computing. Special high speed networks with fibre optic cables and powerful network interface processors are required for building a HC system network. Few examples of HC network systems are Nectar [18], and HiPPI [19].

1.3.6 Programming Environment

Programming environments should provide a layer of abstraction between the user and the machines. This layer should make provision for parallel portable machine-independent programming language. It should provide cross-parallel compilers and cross debuggers that support a wide range of architectures. Linda [12] and mpC [20] are some examples of programming environments for HC systems.

1.3.7 Performance Evaluation

Performance evaluation tools monitor the performance of the HC system. They summarize the runtime behavior of the application, resource usage and determine the cause for a bottleneck. These tools collect, interpret and evaluate the information from various applications, operating systems, network and the hardware in the system.

1.4 Motivation

A static scheduler performs task assignment and scheduling using estimated values. The exact information about the execution times and the communication times are not known when static scheduling is performed. The estimated values may differ from the actual values due to some reasons like congestion in the network or errors in estimation techniques and other fluctuations. However, if the information is available in advance, the static scheduler easily outperforms a dynamic scheduler. Since a dynamic scheduler uses the most recent system information, it is immune to changes in the network properties. Game theory is known to identify equilibrium points in decision making problems, where the agents participating have conflicts in interests. It is also well known that the equilibrium point is a social equilibrium. That is, all the agents are satisfied with their decisions at that point. In HC systems, deciding which machine a subtask is to be assigned so that the entire application benefits is a critical issue. This motivates to us to apply game theory to the problem of task scheduling in HC systems.

1.5 Outline of the Thesis

The outline of the thesis is as follows. Chapter 2 discusses the related works in the area of task assignment and scheduling in a HC system. Chapter 3 gives a brief introduction to the theory of games and auctions. A new framework for dynamic task scheduling using auctions and games is described in chapter 4. Chapter 5 describes the six heuristics that are used to reduce the overhead of the dynamic scheduler. Concluding remarks are presented in Chapter 6. A Game Theoretic Framework for Dynamic Task Scheduling in Distributed Heterogeneous Computing Systems

CHAPTER 2

RELATED WORK

The performance of any HC system depends heavily on how the application is mapped (matched and scheduled) to the system. Therefore, it is important for the scheduling algorithms to be fast, efficient, scalable and able to exploit any unique feature(s) of the system or the application to its advantage. Various algorithms, ranging from simple techniques such as greedy scheduling heuristic to complex graph partitioning and genetic algorithms, have been proposed in the literature. In this chapter, a detailed survey of few mapping techniques for heterogenous computing systems is presented.

2.1 Introduction

An application *program* is a *sequence* of coded procedures and functions held together by some *glue* codes. These procedures and functions can be collectively called as tasks of the application. Some of these tasks may depend on other tasks and therefore have to wait for the inputs before they can start execution, whereas other tasks, which do not depend on any other tasks, may execute without waiting. The data structure that best captures this behaviour of an application is the Task Flow Graph *TFG*. A $TFG(V, E)$ is a directed acyclic graph (*DAG*) in which, every node $v_i \in V$ represents a task in the application and every edge $e_{ij} \in E$ represents dependency of task j on task i. Similarly, the suite of heterogenous processors available in the network can also be viewed as a processor graph, $PG(V, E)$ where, there is a $v_i \in V$ corresponding to every machine in the network and an edge $e_{ij} \in E$ constitutes a communication link from i to j in PG.

Task matching is the process of selecting on *which* processor a particular task should run such that certain optimization criteria are satisfied. Task scheduling follows task mapping and involves determining *when* that task should run on the processor. It has been shown that the mapping problem is NP-complete once the number of processors in the system exceeds 2 [21]. That is, no polynomial time algorithm exists to solve the task mapping problem. Therefore, heuristic-algorithms are needed to solve the mapping problem. Scheduling algorithms have been classified based on the techniques they use to solve the task mapping problem: graph theoretic techniques, list schedulers, optimal selection theory and learning automata to name a few. In the following sections, a brief overview of these techniques is presented. Figure 2.1 illustrates the taxonomy of task assignment algorithms.

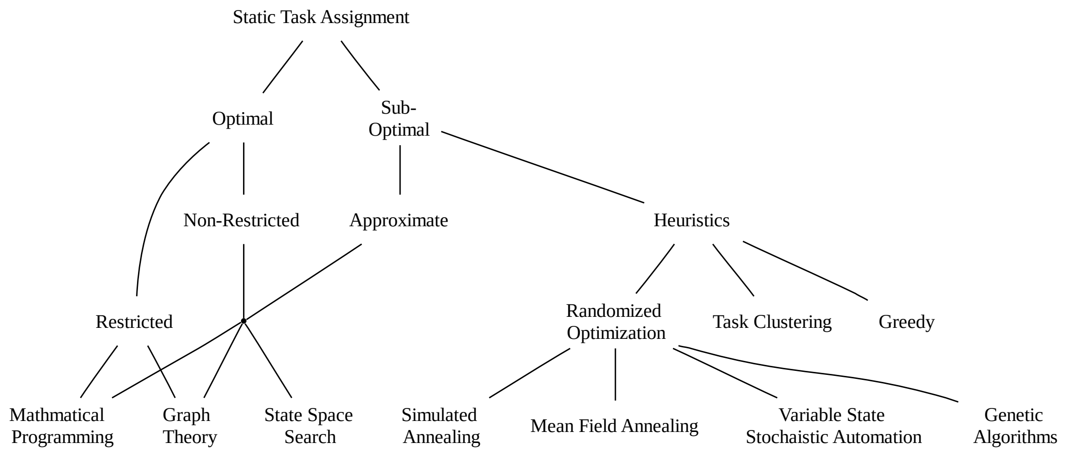


Figure 2.1. Taxonomy of Task Assignment Algorithms [3]

2.2 Graph Theoretic Algorithms

Since applications and networks are represented as graphs, it is intuitive to use graph theoretic techniques to solve the scheduling problem. A popular graph based scheduling is proposed in [21]. The scheduling algorithm is based on max-flow/min-cut heuristic. However, this algorithm can produce optimal solutions only for a two processor system. This work is further extended in [22] to generate sub-optimal solutions for a general HC system. The algorithm called A, uses a heuristic that combines a recursive invocation of max-flow/min-cut algorithm with a greedy type algorithm. It consists of three parts:

1. *Grab:* For an $n-$ processor graph, for each processor p_i a new two-processor graph system is constructed with p_i and $\overline{p_i}$, where $\overline{p_i}$ represents the other $n-1$ processors of the system. The max-flow/min-cut algorithm as outlined in [22] is then used to determine the tasks that can be assigned to processor p_i. This step is repeated for all the processors in the system to generate a partial mapping solution.

2. *Lump:* In this step, all the tasks that remain unmapped in the *grab* step are mapped to one processor.

3. *Greedy:* For those tasks that are unmapped in the *lump* step, tasks with high communication costs between them are identified. All the tasks in the same cluster are then mapped to the processor that can complete their execution at the earliest.

The A* Algorithm Scheduling algorithms based on the A* search technique from the field of artificial intelligence is proposed in [3]. The A* algorithm is used to search efficiently in the search space, in this case, a tree. It searches from a node in the tree known as the *start node*. The intermediate nodes represent the partial solutions and the leaf nodes represent complete solutions or *goals*. Associated with each node is a cost which is computed by a cost function f. The nodes are ordered for search according to this cost, that is, the node with the minimum cost is searched first. Essentially, A* algorithm is a *best first* search algorithm. The value of f for each node n is computed as:

$$f(n) = g(n) + h(n) \tag{2.1}$$

where $g(n)$ is the cost of the search path from the start node to the current node n; $h(n)$ is a lower bound estimate of the path cost from node n to the goal node. Expansion of a node is to generate all of its successors and compute the cost f for each of them. The algorithm maintains a sorted list of nodes and always selects a node with the best cost from this list. The efficiency of this algorithm depends on the accuracy of the prediction of the values for $h(n)$. This is a major drawback of this approach. Moreover, special tree-searching and pruning techniques are required when the number of tasks and machines increase. Another graph theoretic algorithm is proposed in [23] called cluster-M mapping algorithm for mapping tasks with non-uniform computation and communication weights. Two clustering algorithms are proposed which is used to obtain a multilayer clustered graph of tasks (*Spec-graph*) and machines(*Rep-graph*). The cluster-M algorithm is used to map the nodes of spec-graph to nodes of rep-graph. Scheduling algorithm based on the minimum spanning tree of a graph is presented in [24]. The technique exploits the data distribution properties of the application. Two kinds of data distribution are considered in this approach: *data reuse* and *multiple data copies*. Data reuse refers to the condition when two or more subtasks located at the same processor need the same data item from a subtask at another processor. When the tasks reside in different processors and require the same data from a subtask residing in another processor, the condition is referred to as multiple data copies. The algorithm first constructs a TFG from the input application. Then Prim's minimum spanning tree algorithm [25] is used to construct the minimum spanning tree (*MST*) of the TFG. The order in which the vertices of the TFG are added to the MST correspond to the execution order of the subtasks in the application.

Two algorithms are proposed in [26] namely: HP Greedy and the OLROG. The HP Greedy algorithm can be outlined as follows:

1. Partition the TFG into independent subgraphs

2. For each of these subgraphs, beginning from the top of the graph, sort the tasks in each of them based on the weights of the vertices

3. Begin with the heaviest node in the subgraph and assign it to the processor that provides the best expected execution time for that task.

4. Remove the processor to which a task was assigned to and continue the previous step. If the processor list becomes empty, reset it to include all the processors.

The One Level Reach-Out Greedy (*OLROG*) algorithm differs from HP Greedy by taking into account the waiting time when choosing the processor.

2.3 Non Iterative Heuristic Techniques

The non iterative heuristics that have been proposed tend to exploit certain characteristic of the system to provide optimal solution. Opportunistic Load Balancing (OLB) assigns each task, in arbitrary order, to the next available machine, regardless of the task's expected execution time on the machine [27, 28, 29]. User Defined Assignment (UDA) assigns each task, in an arbitrary order, to the machine with the best expected execution time for that task, regardless of the machine's availability [27]. The algorithm is also referred as the Limited Best Assignment (LBA) [27, 28]. Fast greedy assigns each task, in arbitrary order, to the machine with the minimum completion time for the task [27]. The Min-min heuristic begins with the set $\underline{U}$ of all unmapped tasks. Then the set of minimum completion times, $\underline{M} = \{m_i : m_i = \text{min}\ _{0 \leq j < m}(ct(i,j))$, for each $i \in U\}$, is found. Next, the task with the overall *minimum* completion time from M is selected and assigned to the corresponding machine. Hence the heuristic is named *Min-min*. Lastly, the newly mapped task is removed from U, and the process repeats until all tasks are mapped (i.e. $U = \emptyset$) [27, 28, 29]. Intuitively, Min-min attempts to map as many tasks as possible to their first choice of machine, on the basis of completion time, under the assumption that this will result in a shorter makespan. The Max-min heuristic is very similar to Min-min. The Max-min heuristic also begins with the set $\underline{U}$ of all unmapped tasks. Then, the set of minimum completion times, $M = \{m_i : m_i = \text{min}\ _{0 \leq j < m}(ct(i,j))$ for each $i \in U\}$, is found. Next, the task with the overall *maximum* completion time from M is selected and assigned to corresponding machine. Hence the name Max-min. Lastly, the newly mapped task is removed from U, and the process repeats until all the tasks are mapped, i.e.

$U = \emptyset$ [27, 28, 29]. The motivation for this heuristic is to attempt to minimize the penalties incurred by delaying the scheduling of long-running tasks. The assumption here is that with Max-min the tasks with shorter execution times can be mixed with tasks with longer execution times and evenly distributed among the machines, resulting in better machine utilization and a better makespan. Segmented min-min algorithm [30] is an extension to the min-min heuristic discussed earlier. The algorithm sorts the tasks according to the cts. The tasks can sorted into ordered list by the average ct, the minimum ct, or the maximum ct. Then, the task list is partitioned into segments with equal size. The segment of larger tasks is scheduled first and segment of smaller tasks last. For each segment, Min-min is applied to assign tasks to machines. The algorithm is described as follows:

1. Compute the sorting key for each task

   POLICY 1– Smm-avg: Compute the average value of each row in ct matrix

   $$key_i = \sum_j \frac{ct(i,j)}{m}$$

   POLICY 2– Smm-min: Compute the minimum value of each row in the ct matrix

   $$key_i = min_j ct(i,j)$$

   POLICY 3– Smm-max: Compute the maximum value of each row in the ct matrix

   $$key_i = max_j ct(i,j)$$

2. Sort the tasks into a task list in decreasing order of their keys.

3. Partition the tasks evenly into N segments.

4. Schedule each segment in order by applying *Min-min*

The Greedy heuristic is literally a combination on Min-min and Max-min heuristics. The Greedy heuristic applies both Min-min and Max-min and uses the better solution [27, 28]. The heuristics discussed in this section are based on the expected completion time $ct(i, j)$ of a task i on a machine j. A major drawback here is that they fail to consider the communication bandwidth in the network.

2.4 Optimal Selection Theory

Optimal Selection Theory (*OST*) is based on mathematical programming formulation for generating optimal scheduling schemes[1, 17].

In the OST model, the application is decomposed into a set of non-overlapping *code segments* that are totally ordered in time. Each code segment is further partitioned into *code blocks* and are executed on various machines of the same type concurrently. Since the code segments are non overlapping and totally ordered, the completion time of the application equals the sum of the execution times of all the code segments. The OST model makes two assumptions:

1. linear speedup when a code segment runs on multiple copies of a machine type

2. there are sufficient number of machines of each type available

Integer linear programming techniques [1] can be used to minimize the execution time.

The OST model does not consider the communication constraints in the network. Moreover, the second assumption is not always true in most practical situations. Augmented Optimal Selection Theory (*AOST*) [31], augments the OST framework by incorporating the performance of the code segments. It overcomes the drawbacks in OST by

1. It considers the execution time of code segments on all machines

2. allowing non-uniform decomposition of code segments

3. limited number of processors

These assumptions renders AOST more practical than OST. Heterogenous Optimal Selection Theory (*HOST*) [32] is an extension of the AOST framework. It incorporates the effects of various local mapping techniques and allows for the concurrent executions of mutually independent code segments on different machine types. Here, the tasks of the application are divided into sub-tasks. Each subtask consists of a set of code segments which are executed serially. The rest of the formulation of HOST is similar to AOST. Generalized Optimal Selection Theory (*GOST*) [33], extends the selection theory formulations by including tasks modeled by general dependency graphs. The basic code element is a process that corresponds to a code block or a non-decomposable code segment. Therefore, an application consists of a number of processes that can be modeled as a dependency graph. The model assumes that there are τ different types of machines and there are an unlimited number of machines of each type. The objective of the formulation is to generate an optimal mapping such that each node in the dependency graph is assigned to a machine of a particular type. Polynomial time heuristics can then be used to minimize the completion time of the application.

2.5 Simulated Annealing

Simulated annealing is an iterative algorithm that is used to solve many NP-hard and NP-complete problems. Because of its conceptual simplicity and versatility it is used as a tool in a wide area of engineering applications including mapping and scheduling. The algorithm is derived from a process used in metallurgy to make alloys. The core of the algorithm is the *energy function*. The key control parameter in the function is the *temperature* variable. The algorithm initially accepts poor uphill moves, that is, when temperature is high. It does so to avoid being stuck at local minima. But as temperature decreases in the following iterations, the probability of accepting bad solutions decrease. A general simulated annealing algorithm can be outlined as follows:

```
1. Get a random initial solution $S$
2. Set initial temperature $T > 0$
3. While stop criteria not met do
    (a) Perform the following steps $L$ times
        i. Let $S'$ be a random neighbor of $S$
        ii. Let $\Delta$ = cost($S$) - cost($S'$)
        iii. If $\Delta \geq 0$ : Set $S$ = $S'$ $\cdots$ [downhill move]
        iv. If $\Delta < 0$ : Set $S$ = $S'$ with probability $e^{\Delta T}$ $\cdots$ [uphill move]
    (b) Reduce the temperature $T$.
4. Return the best $S$ visited
```

Since the efficiency of algorithm depends on the choice of initial solution and initial temperature, care must be taken to fix these values. Many algorithms for mapping tasks in a heterogenous system are presented in [34, 35, 36]

2.6 Tabu Search

Tabu search is an exhaustive state space search algorithm [37, 38]. The algorithm keeps track of the solutions visited so that it does not visit it again in further searches. The algorithm utilizes a control parameter t known as the length of the table list used. During each iteration, the algorithm exhaustively searches for neighborhood of the current solution not visited within the last 't' iterations. The current solution is then replaced with the neighboring solution with the best cost. The generalized search algorithm can be outlined as:

1. Get a random solution S
2. While stop condition not met
   (a) Let S' be a neighboring solution of S with best Δ = cost(S) - cost(S') and not previously visited in the last t iterations
   (b) Set $S = S'$
3. Return the best S visited

2.7 Genetic Algorithms

Genetic algorithms(*GA*) [39, 40] are generally used for solving problems with huge search space. In general the organization of GA can be outlined as follows:

1. Generate initial population
2. Evaluate the fitness of each chromosome in the population
3. While stopping criteria not met do
   (a) Selection
   (b) Crossover
   (c) Mutation
   (d) Evaluation

The initial population is generated using a uniform random generator or by using a simple heuristic mapping algorithm. Then each chromosome in the population is evaluated for 'fitness'. Fitness function reflect a certain system characteristic like total scheduling cost or total executing time. Following this step, the selection process is performed. The aim of selection process is to quickly prune out poor solutions from the population and to promote good solutions into pool of chromosomes. There are several selection algorithms available in the literature, for example the roulette wheel [41], fitness ranking [42], tournament [43] and stochastic methods [44].

The objective of crossover step is to allow mixing of fit chromosomes from the population to produce super-fit chromosomes. There are several algorithms to achieve the crossover step [45]. The amount of crossover is controlled by a control variable known as probability of crossover. The mutation step is used to avoid getting stuck at local minima by introducing previously discarded bad chromosomes into the population. After mutation, the population is again evaluated for fitness. These four steps are repeated until the stopping criteria is met. In [46] a hybrid GA based algorithm for task scheduling in a multi-processor system is presented. Maheswaran et al. [47] have compared eleven heuristics for task mapping, including greedy, min-min, tabu, simulated annealing and genetic algorithms, and have observed that genetic algorithms produce the best solution.

2.8 Genetic Simulated Annealing Algorithms

The Genetic Simulated Annealing (*GSA*) heuristic is a combination of the GA and SA techniques [48, 49]. In general, GSA follows procedures similar to the GA outlined earlier. GSA operates on a population generated by simple heuristics. It performs similar mutation and crossover operations. However, for the selection process, GSA uses the SA cooling schedule and system temperature, and a simplified SA decision process for accepting or rejecting new chromosomes. GSA uses elitism to guarantee that the best solutions always remained in the population. In [47] the initial population of 200 chromosomes is generated using min-min heuristic. The initial temperature is set to the average makespan of the initial population and decreased by 10% in every iteration. When new chromosome is compared with the corresponding original chromosome (after crossover and/or mutation). The new chromosome is accepted if the new makespan is less than the sum of old makespan and the system temperature. That is, new makespan $<$ (old makespan + temperature)

is true, the new chromosome becomes part of the population. Otherwise, the original chromosome survives to the next iteration. Therefore as the system temperature decreases, it is again more difficult for poorer solutions to be accepted.

Since it uses a probabilistic procedure during selection process, it accepts poor quality intermediate solutions. These poor solutions sometimes do not lead to better final solutions.

2.9 Learning Automata Algorithms

In [2], the author proposes a scheduling algorithm based on learning automata. The model is based on a P-model variable structure stochastic automation (*VSSA*) for optimizing a single cost-metric and a S-model VSSA for optimizing multiple cost-metrics. The VSSA initiates a random action for which the system reacts with a stochastically related response. The VSSA observes this response from the system and re-evaluates the action probabilities using a reinforcement scheme. It perform these operation iteratively to improve the performance of the automata. Essentially, the model 'learns' from the response from the system and adapts itself to choose the best action which optimizes a cost function. Three new cost-metrics that can be used to model the cost function is presented in [50].

The VSSA is represented as a 3-tuple $\langle \underline{\alpha}, \underline{\beta}, A \rangle$ where:

- $\underline{\alpha}$ = output of the automation
- $\underline{\beta}$ = input to the automation
- A = the reinforcement algorithm

The model maintains a separate VSSA for each task $s_i \in TFG$. An action corresponds to the process of assigning a task to a processor. Since a task can be assigned to only a single machine, if there are M machines in PG then the action set $\underline{\alpha}$ contains M actions. The general structure of the learning automata can be outlined as follows:

```
1. Learning_Algorithm:
   (a) While stop criteria not met do:
       i. Generate a solution
       ii. Compute the cost of the generated solution
       iii. If the cost is better than that of previous solution:
           A. Set $E_{resp}$ as favourable response
       iv. else:
           A. Set $E_{resp}$ as an unfavorable response
       v. Translate the response $E_{resp}$ with a heuristic
       vi. Update the action probabilities
```

The crucial steps of the algorithm are steps 1(a)v and 1(a)vi. The input to the automation, β, is determined from the response from the environment,E_{resp} . Six heuristics are proposed in [2] to translate this response. Another important issue is to determine the function that updates the action probabilities. This is done in step 1(a)vi. The updating function comprises of two functions namely, the reward, a and the penalty, b. These functions control the speed of convergence of the algorithm and also the quality of the solution.

The details of extending this algorithm to handle multiple cost-metrics is discussed in [51]. A P-model VSSA can be used to optimize the weighted average of all the cost-functions. Another more complex approach to solve the problem is to use a S-model VSSA that optimizes the individual cost-functions and the responses for the costs are combined to decide the value for $beta$. Studies in [2] show that this latter approach produces better solutions than the weighted average approach.

2.10 List Scheduling Algorithms

The general list scheduling algorithm can be outlined as follows:

1. While there are tasks to be scheduled do:
   (a) Maintain a list of tasks, sorted by their priority
   (b) Task selection
   (c) Processor selection

List scheduling algorithms can be classified into two categories: *static* [52, 53, 4, 54] and *dynamic* [55, 56, 57] algorithms. In static list scheduling algorithms, the tasks are scheduled in the order of their previously computed priorities. A task is usually scheduled on a processor that gives the earliest start time for the given task. Thus, during each scheduling step, first the task is selected and then its destination processor is selected.

Fast Critical Path (*FCP*) [54], reduces the task selection complexity by restricting the choice for the destination processor from all the processors to only *two* processors:

- the task's enabling processor and
- the processor which becomes idle the earliest

The Heterogenous Earliest Finish Time (*HEFT*) [4] algorithm is a DAG scheduling algorithm that supports a bounded number of heterogeneous processing elements (PEs). To set priority to a task t_i, the HEFT algorithm uses the upward rank value, which is defined as the length of the longest path from t_i to the exit node. The rank of a node is determined based on its computation and communication costs. The task list is generated by sorting the nodes with respect to decreasing order of rank values. The algorithm uses earliest finish time, EFT, to select the processor for each task. The running time of HEFT is $O(V^2 \times P)$ where, V is the number of tasks and P is the number of processors in the system.

The Critical-Path-on-a-Processor (*CPOP*)[4] algorithm, is another heuristic for scheduling tasks on a bounded number of heterogenous processors. Critical path is defined as the longest path from the source node to the exit node. All the nodes in the critical path are the critical path nodes. The algorithm evaluates the ranks based on communication and computation costs. The critical path nodes are determined in the next step. The algorithm then identifies the critical path processor (the processor that minimizes the length of the critical path). The CPOP uses the ranks to assign node priority. The processor selection phase has two options:

1. If the current node is on the critical path, it is assigned to a critical path processor (*CPP*)
2. otherwise it is assigned to a processor that minimizes the execution completion time.

The Dynamic-Level Scheduling (*DLS*) [58] algorithm assigns priorities by using an attribute called *Dynamic Level* (*DL*). In contrast to mean values, *median* values are used to compute the static upwards rank; and for earliest start time computation, the non-insertion method is used. At each step, the algorithm selects the $\langle$ready node, available processor $\rangle$ pair that maximizes the DL value. For heterogenous environments $\Delta(n_i, p_j)$ term is added to the DL computation. The Δ value for a task n_i on a processor p_j is computed by the difference between the task's median execution time on all processors and its execution time on the current processor. Levelized-Min Time (*LMT*) [59] algorithm is a two phase algorithm. The first phase orders the tasks based on their precedence constraints, i.e., level by level. This phase groups the tasks that can be executed in parallel. The second phase is a greedy method that assigns each task (level by level) to the ”fastest” available processor as much as possible. A task in a lower level has higher priority for scheduling than a node in a higher level; within the same level, the task with the highest computation cost has the highest priority. If the number of tasks in a level is greater than the number of processors, the fine-grain tasks are merged into a coarse-grain task until the number of tasks equal the number of processors. Then the tasks are sorted in reverse order based on average computation time. Beginning from the largest task, each task will be assigned a processor:

1. that minimizes the sum of the communication costs with tasks in the previous layers

2. that does not have any scheduled task at the same level

The Mapping heuristic (*MH*) [60], uses static upward ranks to assign priorities to the nodes. A ready node list is kept sorted according to the decreasing order of priorities. With a non-insertion based method, the processor that provides the minimum earliest finish time of a task is selected to run the task. After a task is scheduled, the immediate successors of the task are inserted into the list. These steps are repeated until all nodes are scheduled.

In dynamic scheduling algorithms, the tasks do not have pre-computed priorities. At each scheduling step, each ready task is tentatively scheduled to each processor, and the best $\langle$ task , processor$\rangle$ is selected. Thus, at each step, the task and the destination processor are selected at the same time.

In Fast Load Balancing (*FLB*) [57], at each iteration of the algorithm, the ready task that can start the earliest is scheduled to the processor on which that start time is achieved.

The hybrid remapper [61] is a dynamic scheduling heuristic based on a centralized policy used to improve the solution obtained by a static scheduler. The hybrid algorithm works in two phases. The first phase of the algorithm executes prior to application execution. The set of subtasks is partitioned into blocks such that the subtasks in a block do not have any data dependencies among them. however the order among the blocks is determined by the data dependencies that are present among the subtasks of the entire application.

The second phase of the hybrid remapper, executed during application run time, involves remapping the subtasks. The remapping of a subtask is performed in an overlapped fashion with the execution of other subtasks. As the execution of the application proceeds, run-time values for some subtask completion times and machine availability times can be obtained. The hybrid remapper attempts to improve the initial matching and scheduling by using the run time information that becomes available during application execution and information that was obtained prior to the execution of the application. The mapping decisions are based on a mixture of run time and expected values.

Both static and dynamic approaches of list-scheduling have their advantages and disadvantages in terms of schedule quality they produce. Static schedulers are more suited for communication-intensive and irregular problems where, selecting important tasks first is crucial. Dynamic schedulers are more suitable for computation-intensive applications with high degree of parallelism, because these algorithms focus on obtaining good processor utilization.

CHAPTER 3

PRELIMINARIES OF GAME THEORY

3.1 Introduction to Game Theory

Game theory is properly a branch of mathematics. It is used to analyze the behaviours of economic agents who have conflicts in interests. The scope of game theory is varied, ranging from analyzing the behaviour of players of a simple Rock–Paper–Scissors game to nations devising military strategy. Game theory, like any other theory in mathematics, consists of some *axioms* and involves proving certain other assertions and theorems to be true assuming that the axioms are true. The theory also contains certain terms that are to be defined precisely with *primitive terms*. These primitive terms are simply accepted as understood and the axioms are assumed to be true.

The principles and basic building blocks of game theory was proposed by von Neumann in 1928 [62] and Nash[63].

Some important definitions of primitive terms are presented in the following sections.

3.2 Basic Definitions

The theory of games is studied in three levels of abstraction. They are

1. Theory of games in extensive form
2. Theory of games in normal form
3. Theory of games in characteristic function form

The fundamental structure behind all the three abstractions is the *game tree*. To represent a game as a game tree, the following details must be specified:

1. The set of *players* who play the game
2. A set of alternatives or *moves* available to the player during his turn
3. A specification of the *information-set* for each player
4. A termination condition
5. A set of *payoffs* for each player for each *outcome* of a game

Players are the entities that compete with each other in the game. They form the nodes in the game tree. The player who plays first is at the root of the tree. The list of moves available to the player at that instant of the game form the branches of the tree. The nodes to which these first-order branches point to are the possible situations that can result from the choices of first player. Second and higher order branches, representing the choices open to the player who is to play next, issue from these nodes. This branching process continues until a situation defined by the termination condition as the *outcome* of the game, is reached. Figure 3.1 shows an example of a game tree. If a player knows exactly the

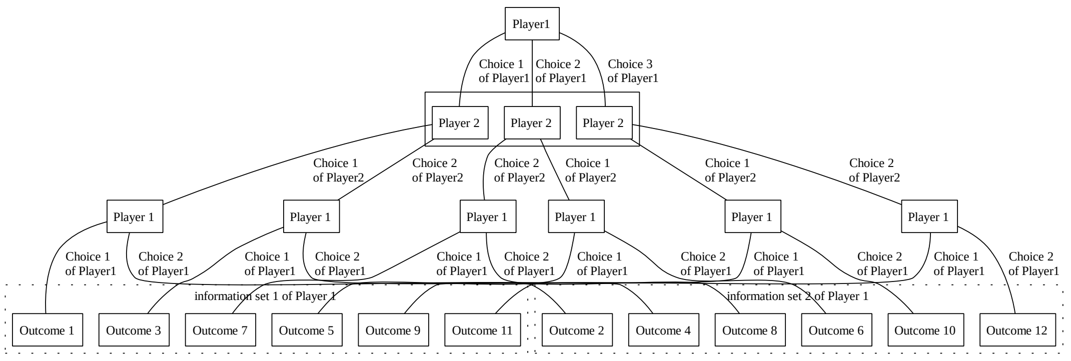


Figure 3.1. An Example of a Game Tree

choices made by the other players who have already moved then he knows to which branch point or node the game has progressed to in the game tree. Such games are known as games of *perfect information*. But if that is not the case, the player can only know the set of nodes to which the game has progressed. This set of nodes is the *information-set* of the player. The gain obtained by the player by making some choices to arrive at an outcome is known as the *payoff* to the player. The most important concept emerging from the analysis of game-tree is that of *strategy*. A strategy is defined as

> *”A strategy is essentially a statement made by a player specifying which of the alternatives he will choose if he finds himself in any of the information sets which are associated with his moves”*[64]

A strategy involves foreseeing *all* the possible situations that may arise in the course of a game. It is shown in game theory that once a strategy is chosen by each player, the outcome of the game is thereby determined.

3.2.1 Games in Extensive Form

Extensive form representation of a game is basically the game-tree itself that captures all of the possible decisions of the game beginning from the root, which is the first move of the first player. The terminal nodes specify the payoffs to each player. This is the lowest level representation of the game where all the internal structures of the strategies of the players are visible.

Figure 3.2 shows the extensive form representation of a simple ”Rock–Paper–Scissors” game played by two players. Rock–Paper–Scissors is a two player game played between two players Player1 and Player2. The moves-set of each player is $\{R, P, S\}$ representing rock, paper and scissors. It is assumed that Player2 plays after Player1 and has perfect information about the moves of Player1. Figure 3.2 shows the extensive form representation for such a game.

Suppose, Player1 and Player2 move simultaneously, Player2 has incomplete information about the exact move of Player1. That is Player2 only knows the subset of nodes to

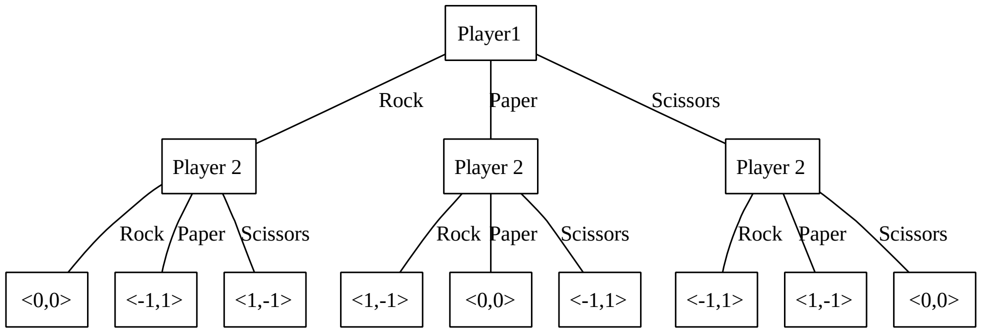


Figure 3.2. Extensive Form Representation of Rock–Paper–Scissors Game

which the game has progressed to. The extensive form representation to illustrate this case is shown in figure 3.3.

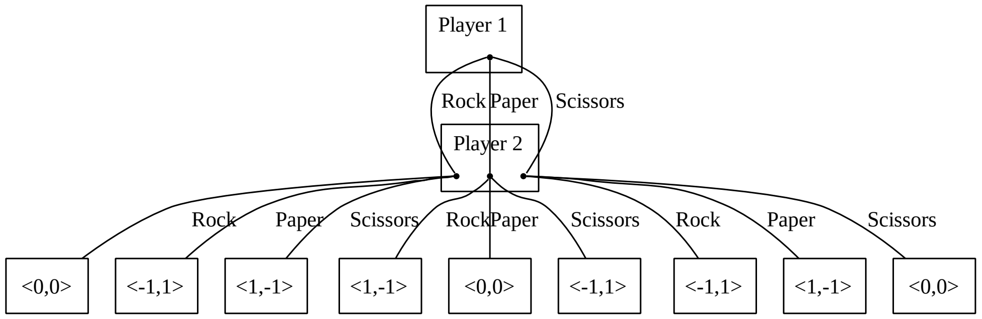


Figure 3.3. Extensive Form Representation when Players Move Simultaneously

3.2.2 Games in Normal Form

Normal form is another popular form of representation of games. Normal form representation hides the internal details of the move-set and deals only with the relation between a strategy of a player and its payoff. It is considered the second level of abstraction while analyzing games. Normal Form Games (*NFG*) are usually represented by multi-dimensional matrices. The normal form representation of the Rock–Paper–Scissors game is shown in table 3.1.

There are two players in the game, therefore the game matrix has two dimensions, one for each player. A player has three moves, that is, either Rock, Paper or Scissors. Hence

Table 3.1. Normal Form Representation of Rock–Paper–Scissors Game

Player1, Player2	Rock	Paper	Scissors
Rock	$0, 0$	$-1, 1$	$1, -1$
Paper	$1, -1$	$0, 0$	$-1, 1$
Scissors	$-1, 1$	$1, -1$	$0, 0$

there are three rows and three columns in the matrix. The payoffs of players when Player1 chooses Rock and Player2 chooses Scissors is stored in location $NFG(Rock, Scissors)$, which is $\langle 1, -1\rangle$ in our case.

3.2.3 Games in Characteristic Function Form

This is the third layer of abstraction used to represent games. Characteristic function games are normally used to study about the co-operation among players and formation of coalitions. The readers are referred to [64] for further details on this class of games.

3.2.4 Types of Games

Games are classified into two major categories, namely: *co-operative* games and *non co-operative* games.

Co-operative Game: A game in which the participants agree to a set of rules and use them to deduce their strategies is a co-operative game.

Non Co-operative Game: A game in which such decisions cannot be made by the participants is a non co-operative game. [63, 65]

In addition to classifying games as co-operative and non co-operative, they are also classified as *zero-sum* games and *non zero-sum* games. The classification of games is represented in figure 3.4.

Zero-Sum game: In this game, the sum of the payoffs of all the players in the game is zero. The example in table 3.1 is an example of a zero-sum game. In these games, whatever one player wins, the other players lose.

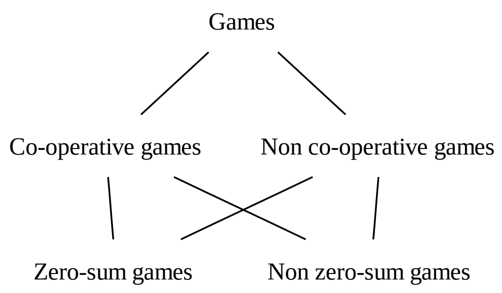


Figure 3.4. Classification of Games

*Non zero-sum game:*In this game, the sum of the payoffs is non-zero. Therefore, more than one player can be declared winner of the game.

3.3 Equilibrium in Games

In 1950, Nash proposed a landmark paper about equilibrium in games [65].An equilibrium represents a solution of the game that is strategically most suited for all the players in the game. There can be more than one equilibrium point for a game.

Nash Equilibrium: If there is a set of strategies with the property that no player can benefit by changing his strategy while others keep their strategies unchanged, then the set of strategies and the corresponding payoff constitute Nash Equilibrium.

Formally: Let the payoffs of n players be $u_i, i \in n$ and the set of possible actions $A = A_1 \times A_2 \times \cdots \times A_n$ be common knowledge to all the players. Let a_{-i} denote the actions of all the players besides player i. Then a Nash Equilibrium is an array of actions $a^* \in A$ such that $u_i(a^*) \geq u_i(a^*_{-i}|a_i)$ for all i and all $a_i \in A$ [65].

Several algorithms have been proposed to find the Nash equilibrium for games. Readers are referred to [66] for further details.

3.4 Auction Theory

Auction theory can be described as the study of buying and selling objects. Some basic terms commonly used is this theory are explained next.

Bid is the highest amount a bidder is willing to pay for an object on sale. *Value* can be described as the metric that a bidder uses to evaluate his bid. If the value of a bidder is known to other bidders, then the auction is known as public auctions. In private auctions, bidders do not have any information about the values of other bidders. *Sale price* is the actual amount to be paid by the winning bidder. *Gain* or payoff is the difference between the bid and the sale price. *Winner's curse* befalls a bidder whose sale price exceeds the value of the object on sale. *Bidding strategies* are the guidelines followed by a bidder to fix a bid price from his value.

3.4.1 Types of Auctions

Based on the policy used for evaluating and accepting bids from bidders, auctions are classified as first-price auctions and second-price auctions. If the bidders are ignorant of the bids of other bidders, then such auctions are known as sealed-bid auctions. In a first-price auction, the winner pays an amount that directly corresponds to his value. Whereas in second-price auction, the winning bidder pays an amount that corresponds to the value of the second highest bidder. The winner of the auction is the bidder with the highest bid.

Based on the number of objects on sale, auctions are classified as single-object auctions and multi-object auctions. Multi-object auctions are further studied as auctions of identical objects and auctions of dissimilar objects.

3.4.2 Game Theory as an Optimization Tool for Task Scheduling

In a system with a number of autonomous components such as processors in heterogeneous computing system, though centralized optimization provides opportunity for efficient optimization, the co-ordination and the transfer of information among components are costly and often infeasible. Hence it is important to develop decentralized optimization schemes which permit the individual components take control of the actions that contribute towards the optimization of a global performance criteria. The motivation for using game theory for scheduling tasks is driven by the fact that decentralized optimization in game theory is achieved by the agents (players of the game) acting selfishly.

CHAPTER 4
DYNAMIC SCHEDULING USING GAME THEORY

A typical HC environment would consist of a heterogeneous suite of machines such as SIMD, MIMD, Dataflow etc., interconnected by a high-speed network. The applications are executed by matching the various computational requirements of the application with the capabilities of the machines. In order to make HC viable several issues need to be addressed. A brief introduction to these issues was presented in the introduction chapter. *Task assignment* and *task scheduling* are considered the most crucial amongst these issues. Collectively they are known as *task mapping*. This work presents a new mapping algorithm based on auctions modeled as games. The chapter is organized as follows. First the problem statement is presented. This is followed by a brief description of the HC model and the dynamic scheduling technique. Finally the results of comparing the proposed scheduler with other schedulers are presented.

4.1 Problem Statement

For the following discussions, it is assumed that the application has been partitioned and profiled. Also, the machines in the HC network have been benchmarked.The application is represented as a Task Flow Graph (TFG). The nodes in the graph corresponds to a subtask of the application. The edges between the nodes correspond to data-dependencies between them. A subtask is a code segment in the application that cannot be further partitioned, and it has to be executed as a single unit in a machine. The HC suite is represented as a Processor Graph (PG) where the nodes of the graph represent the corresponding machines and the edges between them, the interconnection between them. The edges in both the graphs are weighted edges. The edge weight of an edge in the TFG represents the number of data units being transferred. Whereas those in the PG represent the time taken to transfer a single data unit.

The task assignment problem involves determining the machines on which the various subtasks of the application need to execute in order to minimize a certain system cost metric. The cost metric could be total completion time , load on the machines or any other characteristic that is unique to the application.

Whenever an application is submitted for execution, the dynamic scheduler is invoked. The dynamic scheduler runs simultaneously in parallel with the tasks throughout the makespan of the application. From the current system state, the subset of subtasks that are ready to be scheduled and the subset of machines that are available are determined. Then a game theoretic auction model is constructed. Subtasks are therefore assigned based on the outcome of the auction. The system state is then updated after a successful partial assignment.

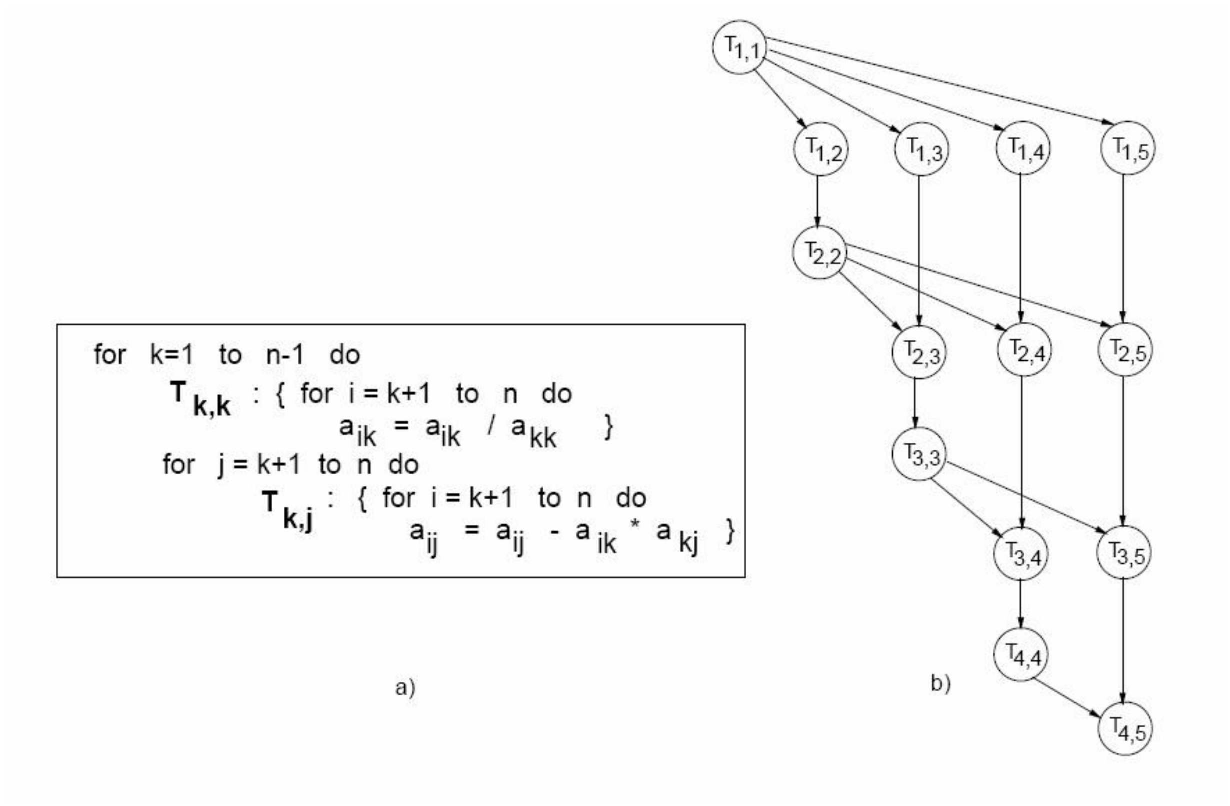
4.2 Elements of the HC System Model

The TFG is constructed by associating every subtask with a node in the graph. For example, the TFG generated for Gaussian elimination algorithm for a matrix of size 5 is shown in Figure 4.1. Another example for a TFG generated for Fast Fourier Transform (FFT) is shown in Figure 4.2.

If S represents the set of subtasks, then

$$S = s_i, 0 \leq i < |S|,$$

which is the set of nodes of a graph with $|S|$ nodes. The data dependencies between the subtasks are represented by a directed edge between the pair of nodes involved. The direction of the edge indicates the direction of data flow. Each edge is assigned a weight



```
for  k=1  to  n-1  do
        T_{k,k} : { for i = k+1  to  n  do
                        a_{ik} = a_{ik} / a_{kk}  }
        for  j = k+1  to n  do
                T_{k,j} : { for i = k+1  to  n  do
                                a_{ij} = a_{ij} - a_{ik} * a_{kj}  }
```

Figure 4.1. The TFG of Gaussian Elimination Algorithm[4]

that corresponds to the number of data units being transferred. It is assumed that these edges do not form any cycles.

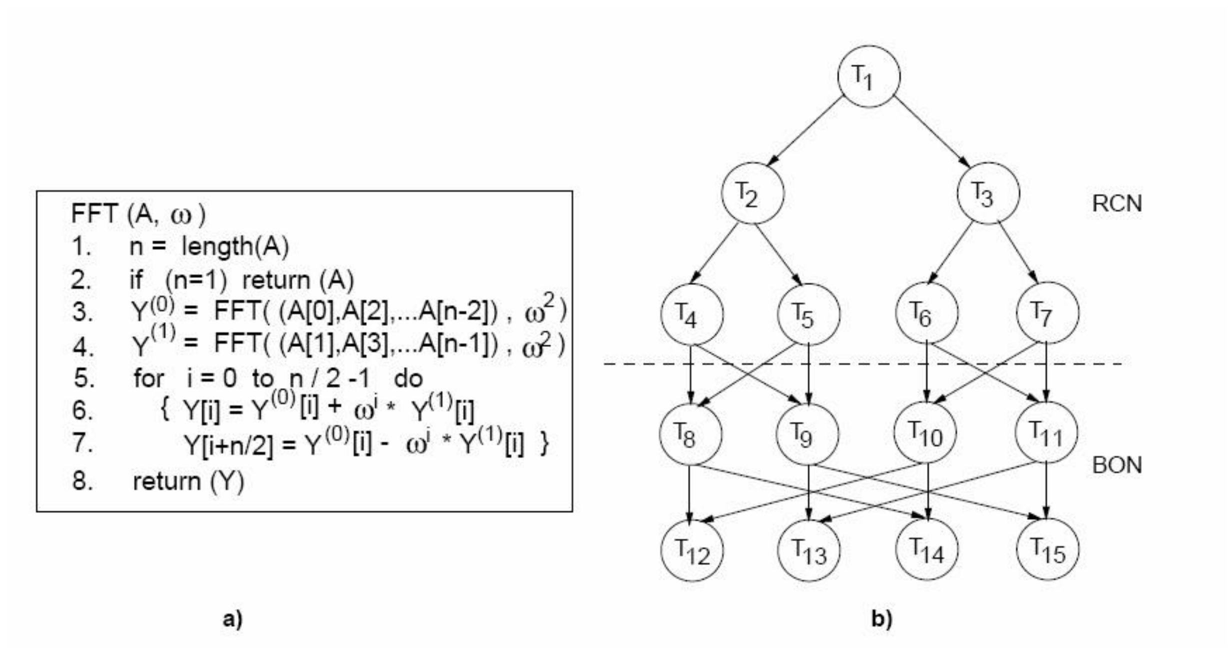
Let E^{TFG} represent the set of edges of the TFG.

$$E^{TFG} = (i,j)/s_i, s_j \in S \text{ \& } s_i \text{ depends on } s_j$$

Let $e_{i,j}^{TFG}$ denote the edge weights. It is defines as

$$e_{i,j}^{TFG} = \begin{cases} \text{number of data units exchanged between } s_i \text{ and } s_j & \text{if } (i,j) \in E^{TFG}; \\ 0 & \text{otherwise} \end{cases}$$

Hence, TFG = (S, E^{TFG}).



```
FFT (A, ω)
1.  n = length(A)
2.  if (n=1) return (A)
3.  $Y^{(0)}$ = FFT( (A[0],A[2],...A[n-2]) , $\omega^2$ )
4.  $Y^{(1)}$ = FFT( (A[1],A[3],...A[n-1]) , $\omega^2$ )
5.  for i = 0 to n / 2 -1 do
6.      { Y[i] = $Y^{(0)}$[i] + $\omega^i$ * $Y^{(1)}$[i]
7.        Y[i+n/2] = $Y^{(0)}$[i] - $\omega^i$ * $Y^{(1)}$[i] }
8.  return (Y)
```

Figure 4.2. The TFG of Fast Fourier Transformation Algorithm[4]

Figure 4.3 illustrates an application program that has been ’atomized’ into ten subtasks. The amount of computation required for each subtask is represented within parentheses. The number of clock cycles required to completely execute the subtask on a baseline machine may be used as a yard stick to quantify the computational requirement of a subtask. For example, subtask 1 in Figure 4.3 requires 327 clock cycles on a baseline machine, whereas subtask 9 requires only 26 clock cycles on the same baseline machine. The edges, as mentioned before, represent the data dependency among subtasks. The dependency of a subtask is quantified as the number of (kilo)bytes of data required by the subtask before it can begin to execute. For example, the edges between subtasks $4, 5, 6$ and subtask 8 in Figure 4.3 implies that subtask 8 depends on subtasks 4, 5 and 6 and it cannot begin to execute before these tasks complete their execution. The edge weights imply that subtask 8 requires 43 KB of data from subtask 6, 26 KB of data from subtask 5 and 13 KB of data from subtask 4. Also, subtask 8 is a dependency to subtask 10, that is, after completion of execution, subtask 8 must supply subtask 10 with 35 KB of data.

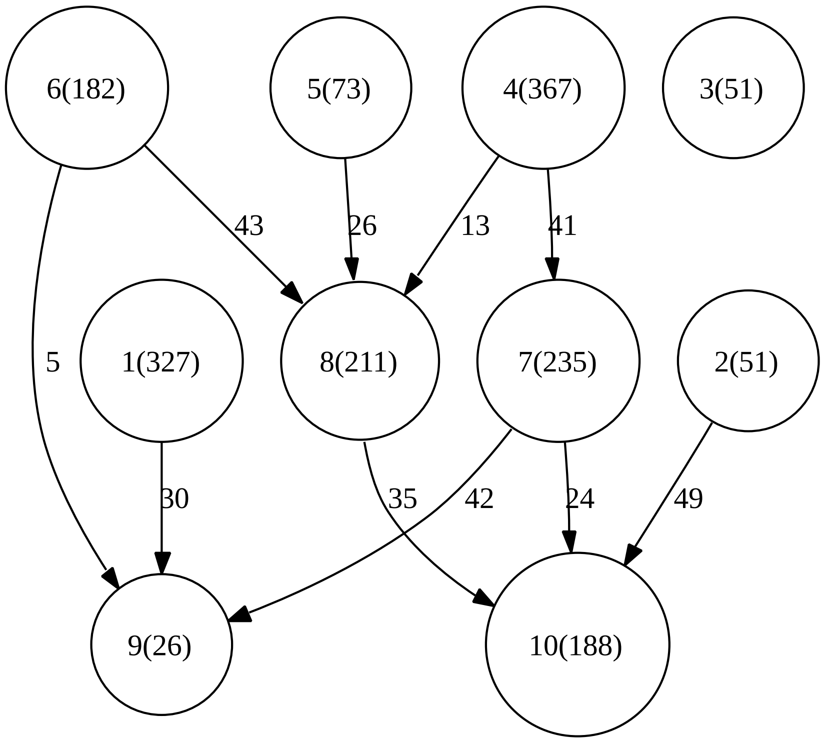


Figure 4.3. An Application Task Flow Graph with $|S| = 10$ and $|E^{TFG}| = 10$

PG is constructed in a similar fashion. Each machine in the system is associated with a node in the PG. If M denotes the set of machines, then

$$M = m_i, 0 \leq i < |M|,$$

which denotes that set of nodes in the PG with $|M|$ nodes. Since the interconnection topology is assumed to be known, an edge is associated with every pair of connected machines. Let E^{PG} represent the set of edges in PG.

$$E^{PG} = (i, j)/m_i, m_j \in M, \text{and } m_i \text{ is connected to } m_j$$

The weight assigned to these edges, $e_{i,j}^{PG}$ are defined as:

$$e_{i,j}^{PG} = \begin{cases} \text{cost of communicating a data unit between } m_i \text{ and } m_j & \text{if } (i,j) \in E^{PG} \\ \infty, \text{ otherwise} \end{cases}$$

Thus $PG = (S, E^{PG})$. Figure 4.4 shows a sample processor graph with $|M| = 3$ and $|E^{PG}| = 3$.

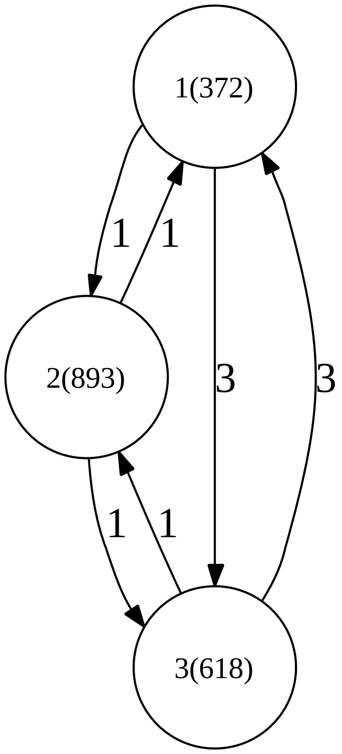


Figure 4.4. An Example of Processor Graph with $|M| = 3$ and $|E^{PG}| = 3$

The solution space for the task assignment and scheduling problem can now be characterized by a mapping of the nodes in the TFG to those of the PG. If i denotes a point in time during the makespan of the problem, then $\pi^s(i)$ represents the initial mapping produced by a static scheduler and $\pi^d(i)$ denotes the partial mapping produced by the dynamic scheduler at that point of time.

$$\pi^s : S \to M$$

$$\pi^d : S'_i \subset S \to M'_i \subset M$$

where S'_i is the subset of subtasks being considered for mapping at time i and M'_i is the subset of machines that are available during that time.

4.2.1 Cost Metric

The objective of task mapping in HC system is to improve the performance of the system for a given application. The cost metric in the system characterizes the quality of the solution. Improvement in performance could be minimization of the total completion time, or minimizing load on the maximum loaded processor or a specific characteristic of an application. The cost metric should reflect the chosen performance criterion of the system. In [2], a detailed description of the various cost metrics are presented. The cost metric used in this work is the total completion time of the application. The system can be easily modified to study other performance criteria. The cost functions are constructed from matrices whose values are obtained from the information from benchmarking and profiling techniques. Actual execution times are used by the dynamic scheduler whenever available. For other situations, the expected execution times are used to create the cost matrix. Let the execution time matrix be denoted by E_T.

$$E\_T = e_t(i,j), 0 \le i < |S|, 0 \le j < |M|$$

$$e\_t(i,j) = \text{execution time of subtask } s_i \text{ on machine } m_j$$

The number of data units exchanged between every pair of connected subtasks are contained in D_X.

$$D\_X = d\_x(i,j), 0 \le i < S, 0 \le j < |S|$$

$$d\_x = e_{i,j}^{TFG}$$

The cost of communicating a single data unit between a pair of connected machines is stored in C_C.

$$C\_C = c\_c(i,j), 0 \leq i < |M|, 0 \leq j < |M|$$

$$c\_c(i,j) = e_{i,j}^{PG}$$

Let *comp(n)* denote the total computation time for a particular assignment and *comm(n)* represent the total communication time at a particular time instant n. Hence:

$$comp(n) = \sum_{i=0}^{|S|-1} e\_t(i, \pi_n(s_i))$$

$$comm(n) = \sum_{i=0}^{|S|-1} \sum_{j=0}^{|S|-1} d\_x(i,j) \times c\_c(\pi_n(i), \pi_n(j))$$

The cost metric completion time would then be

$$Cost(n) = comp(n) + comm(n)$$

Also it is to be noted that

$$\pi_n(i) = \begin{cases} \pi^d(i) & \text{if } i \text{ had been dynamically scheduled by time } n \\ \pi^s(i) & \text{otherwise} \end{cases}$$

4.3 Dynamic Task Scheduling as a First Price Sealed Bid Auction

A group of interested and competitive buyers, a seller and an object of interest constitute an auction. The decision about which buyer gains the object is made by conducting auctions. There are different types of auctions e.g. First Price auction, Second Price auction, Vickery auction etc. The auction that is most suited for dynamic scheduling however is First Price Sealed Bid auction. In this auction, the buyers have independent private value of the object. Bids are placed solely based on this private value. All the buyers are rational. The buyers follow certain strategies to generate the bid. The bid of the buyer is known only to the seller and the buyer himself, hence the name sealed bid. It is well known that the dominant strategy in this type of auction is to bid for an amount that is equal to the private value. The seller then decides the winner based on the bids received. The seller sells the object to the winner for a price that is proportional to the bid.

Typically a dynamic scheduler has to map a subset of subtasks, S'_i to a subset of machines,M'_i, i.e. it has to solve a series of partial mapping problem to achieve a global optimum. This partial mapping problem is modeled as First Price Sealed Bid auction and solved using Game Theory. The subtasks in S'_i are modeled as the buyers in the auction competing to buy machines in M'_i from the seller. The execution times of subtasks in the various machines in the HC suite are considered as private values of the subtasks. A bid consists of a list of machines along with the cost associated with executing the subtask on the corresponding machine. Hence, the First Price Sealed Bid auction can be extended to solve the dynamic scheduling problem.

Figure 4.6 is a space-time representation of the system state when subtask s_{i-1} has started execution in machine m_0. The subtasks that are now ready to be scheduled are subtasks s_i, s_j and s_k. They are the players in the auction. Machines m_1, m_3 and m_4 are the machines to which subtasks si, s_j and s_k have been assigned to respectively by the static scheduler. The following sections provide a detailed description of the dynamic scheduling algorithm.

4.4 Dynamic Scheduling Algorithm

The pseudo-code describing the algorithm is presented in Algorithm 1.

A strategy can be described as a guideline used by a buyer to generate bids. Following this, the buyers in the proposed technique have two strategies. They have been named

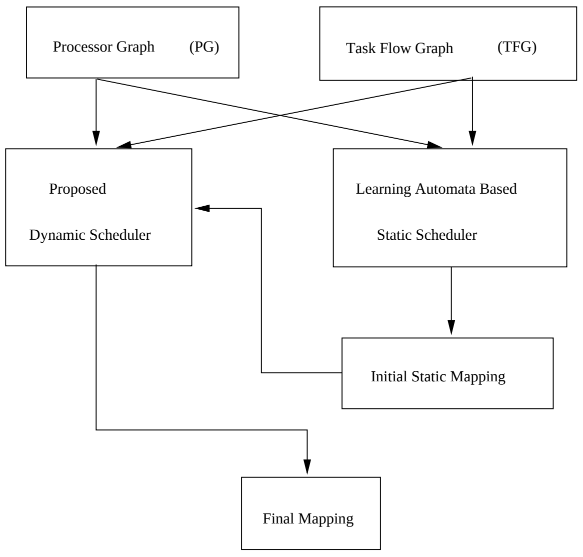


Figure 4.5. Framework of the Proposed Dynamic Scheduler

Conservative and Aggressive Strategies respectively. There are a set of actions or moves associated with each of these strategies. The Nash Equilibrium provides a probability value corresponding to each of these moves. The final mapping is performed by selecting the move corresponding to these probability values.

4.4.1 The Conservative Strategy

This the first of the two strategies used by the buyers to generate bids. There is only one move corresponding to this strategy, i.e. the machine that was assigned by the static scheduler. If a buyer chose only the conservative strategy, the corresponding task would begin execution in the machine that was assigned by the static scheduler at the time determined it. This strategy does not help in searching the solution space. However this strategy is used for the heuristic approach presented in the next chapter. Also, the move corresponding to this strategy leads to selection of a machine that is idle at that instant. Hence this strategy is included in the pure dynamic approach.

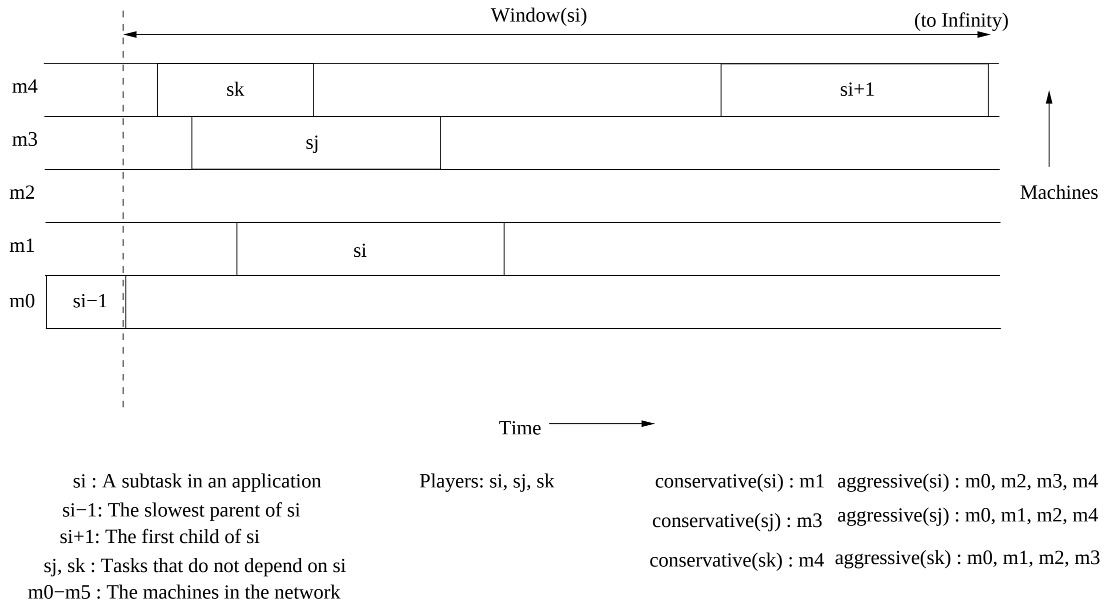

Window(si)
(to Infinity)
m4
m3
m2
m1
m0
sk
si+1
sj
si
si−1
Machines
Time
si : A subtask in an application
si−1: The slowest parent of si
si+1: The first child of si
sj, sk : Tasks that do not depend on si
m0−m5 : The machines in the network
Players: si, sj, sk
conservative(si) : m1
aggressive(si) : m0, m2, m3, m4
conservative(sj) : m3
aggressive(sj) : m0, m1, m2, m4
conservative(sk) : m4
aggressive(sk) : m0, m1, m2, m3

Figure 4.6. Freeze-frame Illustrating the System State when s_i, s_j and s_k are being Dynamically Scheduled using the Proposed Game Theoretic Scheduler

4.4.2 The Aggressive Strategy

The second of the two strategies proposed is used to explore the solution space. When this strategy is chosen, the buyer looks for a machine other than the one assigned to it by the static scheduler. Steps 3 – 10 of Algorithm 1 achieve this goal. The following is a step by step description of how a bid corresponding to this strategy is generated.

- Firstly, a window of time within which a subtask s_i needs to execute is determined. For a pure dynamic scheduler, the lower boundary is set to the current time instant and the upper boundary is set to infinity. Algorithm 2 describes this process.

- In the second step, all the machines m_j that are available during the window of subtask s_i are deemed potential candidates. That is, s_i can execute on m_j at this time instant. A list is maintained for each s_i. This list forms the preference list of the buyer.

- Amongst all the m_j in the list, machines which provide a completion time earlier than $\pi^s(s_i)$ correspond to the list of moves of this strategy.

Algorithm 1 Dynamic Scheduler based on First Price Sealed Bid Auction

```
1: for $s_i \in S'$ do
2:     Set conservative[$s_i$] = $\pi^s(s_i)$
3:     $W_i$ = Window($s_i$)
4:     for $m_j \in M'$ do
5:         if $W_i$ within IdleTime($m_j$) then
6:             Append $m_j$ to the list Candidates[$s_i$].
7:         end if
8:     end for
9: end for
10: Set aggressive[$s_i$] = BestCandidates(Candidates[$s_i$])
11: P = Payoffs($S'$,conservative,aggressive)
12: $\pi^d(\forall s_i \in S')$ = NashEquilibrium($S'$,<conservative, aggressive>,P)
13: Schedule($S'$, $\pi^d(\forall s_i \in S')$)
```

Algorithm 2 Window(s_i)

```
1: Set Window[$s_i$].LowerBound $\leftarrow$ Completion Time of the Slowest Parent
2: Set Window[$s_i$].UpperBound $\leftarrow \infty$
3: Return Window
```

A final list is then complied using the moves possible when using the two strategies. This final list is submitted as the bid of s_i.

4.4.3 Calculation of the Payoff

An outcome of the game is defined as a partial mapping obtained when all the buyers follow a move. The payoff function for an outcome is proportional to the overall completion time of the partial mapping. If a lower completion time is achieved for an outcome, it is rewarded with a higher payoff value for the players.

4.4.4 Dynamic Task Mapping based on Nash Equilibrium

The auction is implemented as an n-person game. The buyers, moves and payoffs constitute an n-person game. Nash Equilibrium is then used to solve the game. The solution of the game is expressed as a probability value corresponding to each move for every player.

Algorithm 3 BestCandidates(List)

```
1: for $s_i \in S'$ do
2:     for $m_j \in$ List do
3:         if Cost($s_i, m_j$) < Cost($s_i, \pi^s(s_i)$) then
4:             Append $\pi^d(s_i)$ to the bid list
5:         end if
6:     end for
7: end for
8: Return the list of bids
```

Dynamic task assignment is hence performed by generating a discrete random variable corresponding to the probability values of the moves.

4.4.5 Analysis of the Algorithm

The worst case complexity of the proposed algorithm depends on the size of the payoff matrix. For a game with N players and S strategies, the size of the payoff matrix is $O(N \cdot S^N)$. Hence, the complexity of the proposed scheduler is dominated by the algorithm used for finding the Nash Equilibrium. It has been shown that *there is a guaranteed equilibrium point* if the moves were allowed to be mixed [63]. Because of a guaranteed solution it is unlikely for the problem to be NP-hard. Also, a proof for the complexity of computing a Nash Equilibrium point to lie between ”P” and ”NP” can be found in [63]. The proposed work uses *Gambit*[66], a software tool for computing Nash Equilibrium.

4.5 Simulation Results and Discussion

The pseudo-code in Algorithm 4 describes the procedure used to simulate the proposed dynamic scheduling technique. The application task flow graph is divided into different levels such that all the tasks within a level are independent and can execute in parallel. When a task from level $i - 1$ begins its execution, the tasks at level i are scheduled. The dynamic scheduler is implemented a batch scheduler that uses a sliding window technique for task scheduling. Though, the best optimization can be achieved by including all the subtasks in level i for scheduling, it would not justify the enormous overhead that would be incurred by the scheduler. Hence the sliding window technique is adopted. A detailed description of the sliding window technique is presented in [67]. Step $5 - 11$ of Algorithm simulate the sliding window technique.

Algorithm 4 Simulation Procedure

```
1: Levels = Levelize(T)
2: for List ∈ tasks in each Level do
3:     Arrange the subtasks in the List in non-decreasing order of their static schedule
       times
4:     while There are unscheduled subtasks in List do
5:         if sizeof(List) ≤ WINDOW_THRESHOLD then
6:             DynamicSchedule(List)
7:         else
8:             Remove the first WINDOW_THRESHOLD values from List and append them
               to BATCH
9:             DynamicSchedule(BATCH)
10:            Clear BATCH
11:        end if
12:    end while
13: end for
```

In order to show the effectiveness of the proposed dynamic scheduler it is compared with a state-of-the-art dynamic scheduling algorithm based on Genetic Algorithm proposed in [68]. The two dynamic schedulers use the solution from the Learning Automata based static scheduler proposed in [2] as an initial solution and try to then improve it. Therefore the static scheduler is also compared with the two schedulers. As yet, a representative set of heterogeneous computing task benchmarks do not exist [69]. Therefore, simulations are performed on randomly generated task flow graphs. The data from Table 4.1 was used to generate the random graphs. As adopted in [2], TFGs were categorized into three major categories depending on the communication complexity. A TFG with number of edges equal to $\frac{|S|}{3}$ are classified as TFGs with low communication complexity. Those with values $\frac{2|S|}{3}$ and $|S|$ are classified as TFGs with medium and high communication complexity. The TFG size was varied from 10 nodes to 100 nodes. The PG size was also varied from 2 machines to a 20 machine system. The graphs were plotted to study the effects of the communication complexity, problem size and network size on the completion time of the application. In order to capture the real-time behaviour, A second set of execution times and communication cost were used by the dynamic algorithm. These values were randomly generated with the expected execution time/communication cost as the mean. Simulations were performed on 10 instances of graphs for each type of the TFG and the averages were used for constructing the graphs.

Table 4.1. Data used for Generating Random Graphs

Number of Tasks $\|S\|$	10, 25, 50, 100
Number of Machines $\|M\|$	2, 5, 10, 20
Number of Edges	$\frac{\|S\|}{3}, \frac{2\|S\|}{3}, \|S\|$
Execution Matrix Data range	1000
Communication Matrix Data range	4
Data Exchange Matrix Data range	500

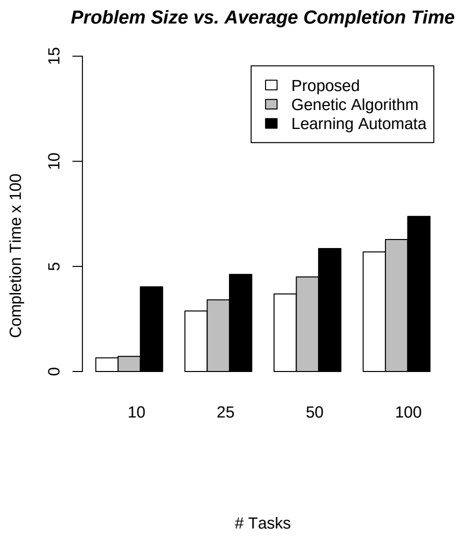


Figure 4.7. Low Communication Complexity , $|M| = 2$

From Figures 4.7 – 4.18 we can observe that the proposed approach provides superior schedules than the other scheduling algorithms for major spectrum of class of TFG. There were some class of TFG e.g. Figures 4.8, 4.11, 4.14 for which Genetic Algorithms produced better results. It is also observed that the proposed scheduling technique was able

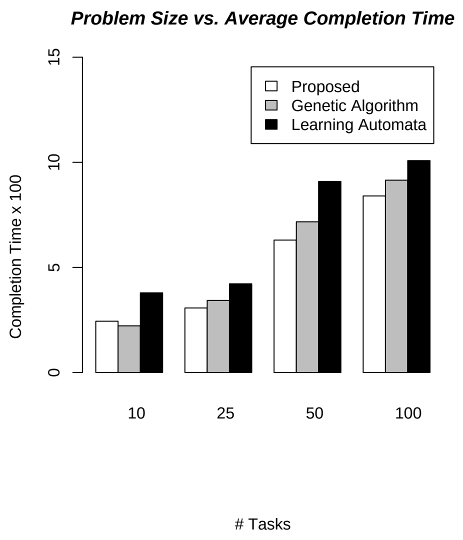


Figure 4.8. Medium Communication Complexity , $|M| = 2$

to consistently improve the static solution by about $20 - 30\%$ for all the cases. Also the proposed game theoretic scheduler provides, on an average, an improvement of 8.8% over the dynamic genetic algorithm.

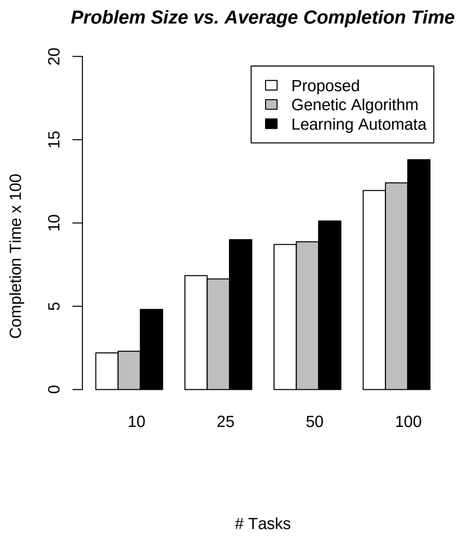


Figure 4.9. High Communication Complexity , $|M| = 2$

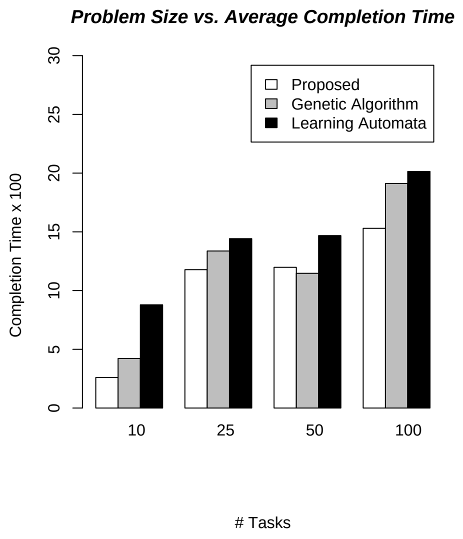


Figure 4.10. Low Communication Complexity , $|M| = 5$

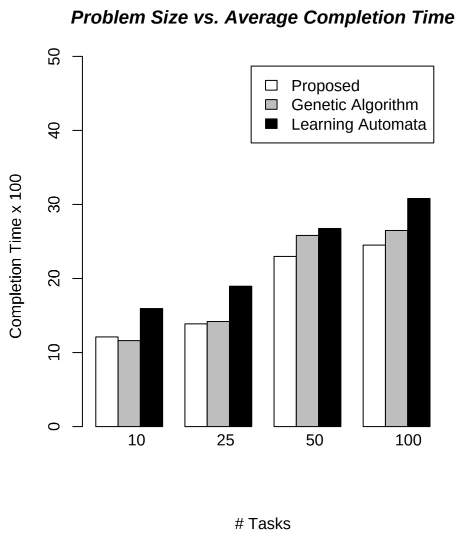


Figure 4.11. Medium Communication Complexity , $|M| = 5$

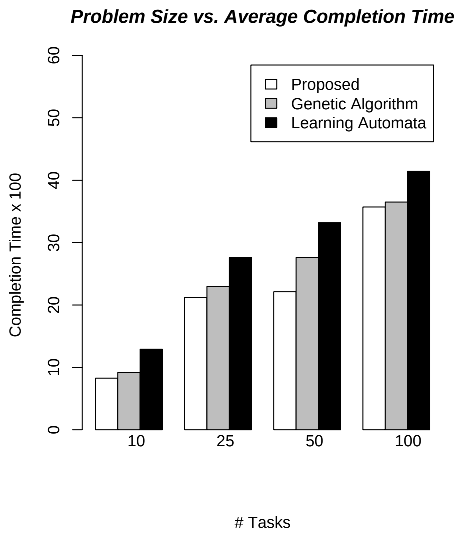


Figure 4.12. High Communication Complexity , $|M| = 5$

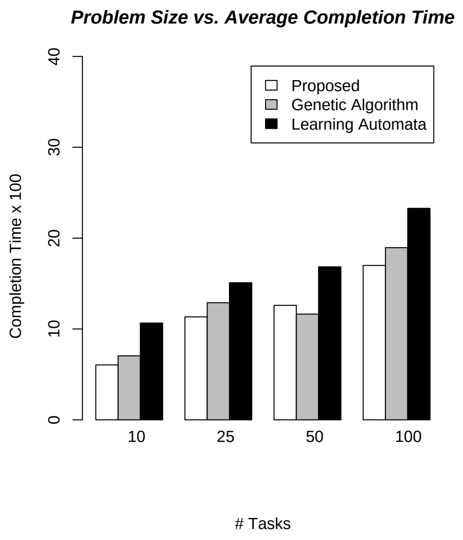


Figure 4.13. Low Communication Complexity , $|M| = 10$

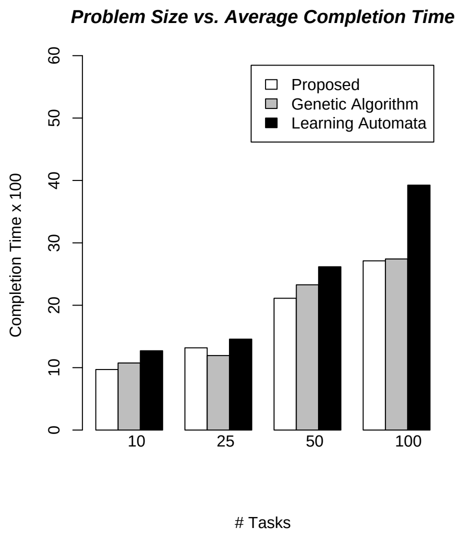


Figure 4.14. Medium Communication Complexity , $|M| = 10$

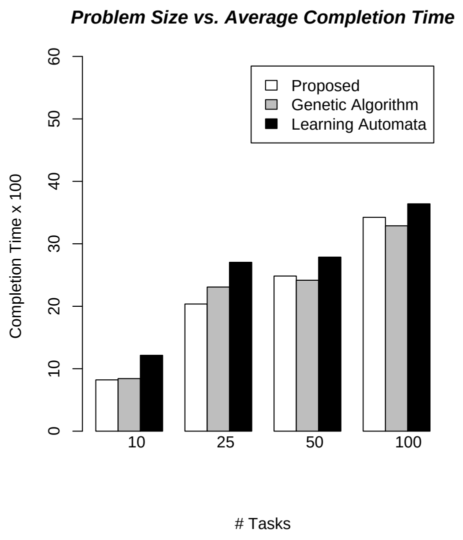


Figure 4.15. High Communication Complexity , $|M| = 10$

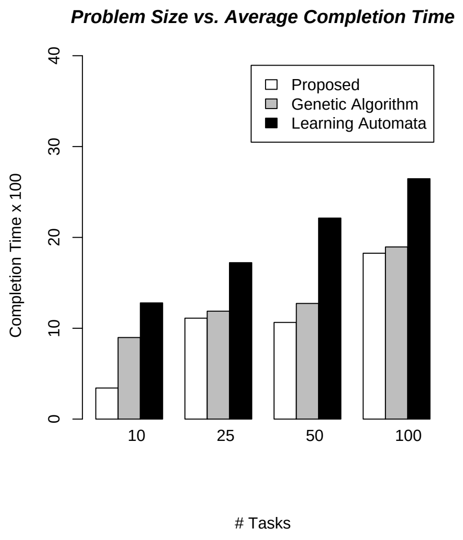


Figure 4.16. Low Communication Complexity , $|M| = 20$

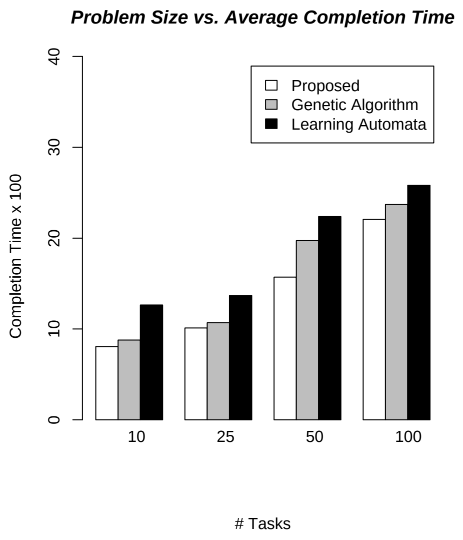


Figure 4.17. Medium Communication Complexity , $|M| = 20$

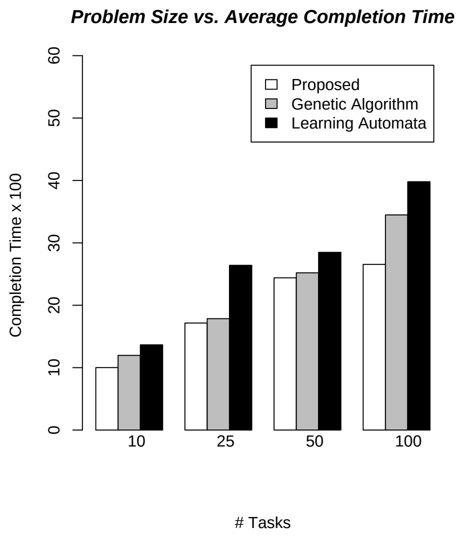


Figure 4.18. High Communication Complexity , $|M| = 20$

CHAPTER 5
DYNAMIC SCHEDULING USING HEURISTICS

5.1 Motivation

In the scheduler proposed in the previous chapter, dynamic scheduler is invoked for all the subtasks. Every time the dynamic scheduler is invoked , it incurs an overhead because some system resources are dedicated to the dynamic scheduler for performing the scheduling operation. In order to reduce the number of times the dynamic scheduler is invoked, a heuristic based approach to select tasks for dynamic scheduler is investigated. Consider a hypothetical scenario in which subtask s_i is scheduled to machine A and subtasks s_j, s_k to machines B and C respectively by the static scheduler. Also assume that s_i can execute only on A whereas s_j and s_k can run on all three machines . If s_i, s_j and s_k fall in to the same time window for scheduling, auction is performed with three buyers. Subtasks s_j and s_k play the game with possibly A included in their moves set. At the end of auction, s_i wins A and s_j and s_k have played the game with an extra move that will never benefit them. One approach to reduce the occurrence of such a scenario would be if s_i were not considered for dynamic scheduling at all. The auction would have one less player and one less move for each player. This could potentially improve the time taken by the dynamic scheduler to complete a partial mapping because the complexity of the scheduler depends on the number of players and number of moves for each player. Selecting subtasks that are to be dynamically scheduled such that the overall completion time of the application is reduced, is NP hard. Therefore, heuristics are used to select the subtasks that are to not be scheduled by the dynamic scheduler, ie. the static schedule is performed on such selected subtasks. Figure 5.1 shows the conceptual block diagram for the proposed heuristic technique. Figure 5.1 illustrates the framework of the proposed heuristic based approach to task scheduling using the game theoretic dynamic scheduler proposed in the previous chapter. The approach is based on the fact that the reduction in the number of subtasks involved in an auction could considerably increase the speed of the dynamic algorithm. The reduction in number of players involved in an auction also restricts the improvement of the solution quality of the dynamic mapping. That is, fewer the number of players involved, the more closer the obtained solution will be to the static mapping. Therefore a trade off must be struck between the solution quality of and the overhead consumed by the dynamic scheduler. This chapter investigates this.

5.2 Heuristic Based Model

The proposed heuristic scheduler comprises of two components. The first component is the Learning Automata based static scheduler proposed in [2]. This component is used prior to the execution of the tasks. The second component consists of the dynamic scheduler proposed in the previous chapter. This component executes concurrently with the execution of the subtasks. The proposed dynamic scheduler attempts to improve the initial static mapping.

As evident from Figure 5.1, there are two additional steps involved in the heuristic based approach. They are used to identify *how many* subtasks need to be dynamically scheduled and *which* subtasks are to be statically scheduled. The former is achieved by defining a parameter μ and the latter is achieved by using heuristics. Once the subtasks are identified, dynamic scheduling is performed only on those subtasks. The following sections present a detailed discussion on how the process of task selection is performed. To determine the number of tasks that need to be statically scheduled , i.e. ignored by the dynamic scheduler, we define a parameter μ called the Mix. The μ-parameter is represented as the ratio of tasks in the application that needs to be ignored by the dynamic scheduler to the total number of subtasks. This parameter enables the heuristic scheduler to behave in a wide spectrum ranging from pure static scheduler with $\mu = 1.0$ to a pure dynamic scheduler with $\mu = 0$

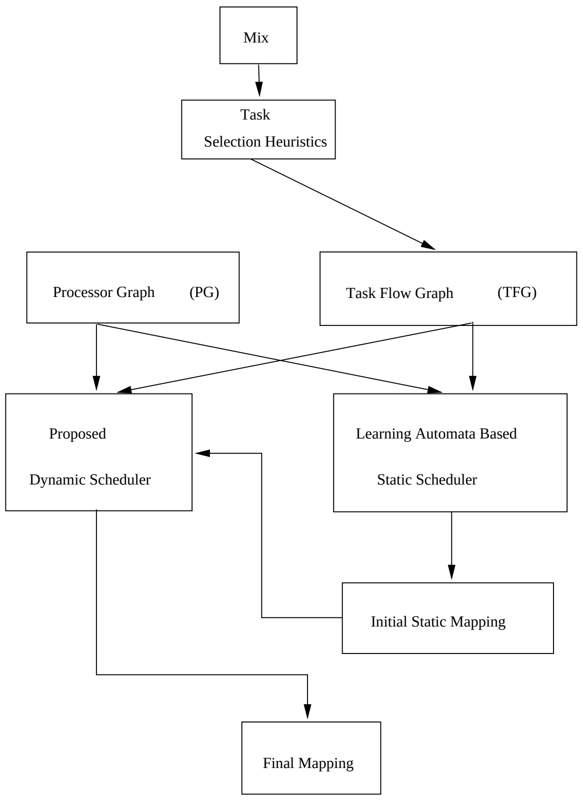


Figure 5.1. Framework of the proposed Hybrid Scheduler

5.3 Selection Heuristics for the Proposed Scheduler

The determination of the subset of nodes to be statically scheduled (T) and those to be dynamically scheduled (Y) to obtain an optimal solution, is an open problem. Therefore heuristics are used to determine these sets. In this work six heuristics are investigated. The details of the heuristics are discussed in the following sections.

5.3.1 Selection Heuristic H1

The critical path of an application provides a lower bound of the makespan of the problem. The first heuristic, called H1, selects the tasks in the critical path of the application. Hence H1 can be formally defined as:

$$H1 \rightarrow \begin{cases} T = T \cup t_i & \text{if } t_i \in \text{ Critical path of TFG} \\ Y = Y \cup t_i & \text{if } t_i \notin \text{ Critical path of TFG} \\ \forall_{i=1}^{n} t_i \in T \end{cases}$$

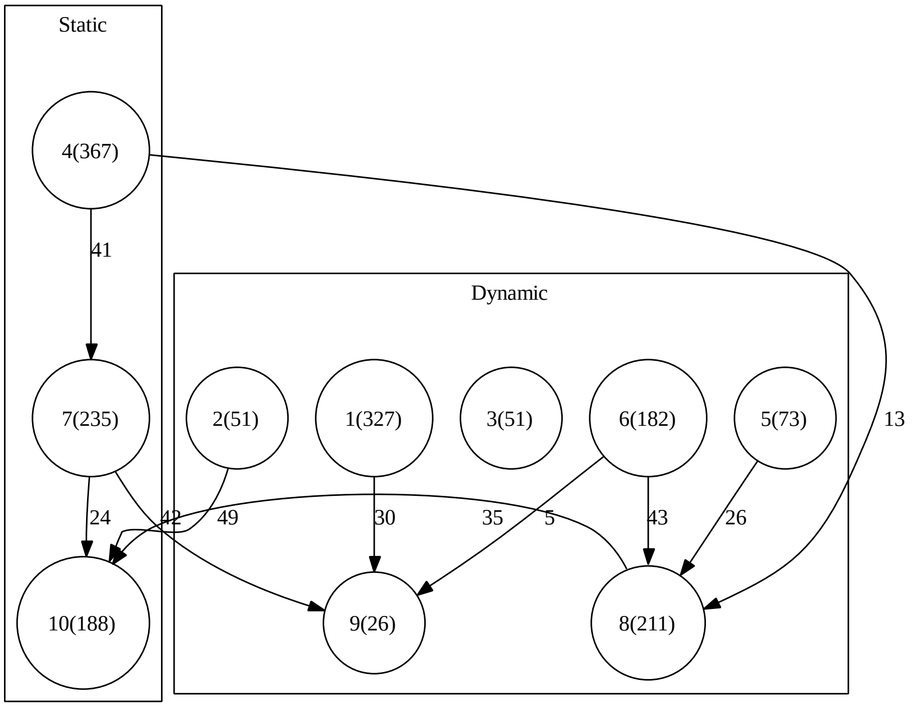


Figure 5.2. The TFG of Figure 4.3 after Application of Selection Heuristic H1

Figure 5.2 depicts the effect of applying this heuristic to the task flow graph of Figure 4.3. The cluster marked *Static* contains the nodes that are to be scheduled statically and the cluster marked *Dynamic* contains the nodes there are to be scheduled dynamically.

Algorithm 5 Pseudo-code for Translation of Selection Heuristic H1

```
1: $p =$ FindCriticalPath($TFG$);
2: $T = \emptyset$;
3: $Y = \emptyset$;
4: for $v \in S$ do
5:     if $v \in p$ then
6:         $T \leftarrow T \cup v$;
7:     else
8:         $Y \leftarrow Y \cup v$;
9:     end if
10: end for
```

5.3.2 Selection Heuristic H2

The second heuristic, H2, simply selects the subtasks at random. That is,

$$H2 \rightarrow \begin{cases} T = S \cup \forall_{i=1}^{k} \text{ random}(S) \\ Y = S - T \end{cases}$$

This heuristic, by far, is the simplest heuristic. It does not take the structure of the TFG into account. The pseudo-code for implementing this heuristic an be outlined in algorithm 6 and the effect after the heuristic is applied is shown in Figure 5.3.

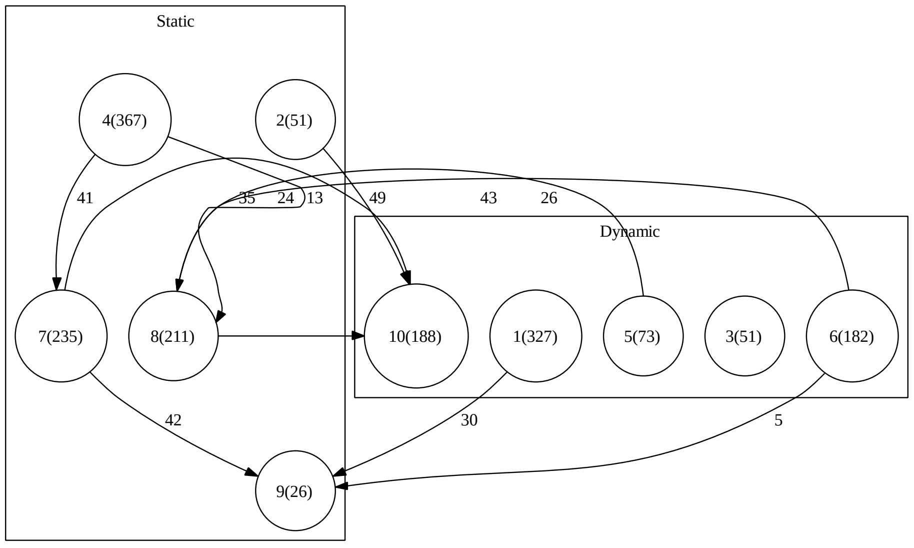


Figure 5.3. The TFG of Figure 4.3 after Application of Selection Heuristic H2

Algorithm 6 Pseudo-code for Translation of Selection Heuristic H2

```
1: $T \leftarrow \emptyset$;
2: $Y \leftarrow \emptyset$;
3: $k \leftarrow$ int($\frac{|S|}{2}$);
4: for $i$ in 1 to $k$ do
5:     $v \leftarrow$ random($S$);
6:     $T \leftarrow T \cup v$;
7: end for
8: $Y \leftarrow S|T$;
```

5.3.3 Selection Heuristic H3

The third heuristic, H3, selects subtasks that have high dependency between them for static scheduling and the remaining subtasks are scheduled dynamically. Mathematically, H3 can be stated as:

$$H3 \rightarrow \begin{cases} T = T \cup \mathrm{head}(e) \cup \mathrm{tail}(e)\ \forall e \in E \quad \text{such that EdgeWeight}(e) \geq W_{th} \\ Y = Y - T \end{cases}$$

Where W_{th} is defined as the threshold weight. It is calculated as

$$W_{th} = \frac{\Sigma_{\forall e \in E} \text{EdgeWeight}(e)}{|E|}$$

The average bandwidth of network can also be added as an additional constraint to evaluate W_{th}. If the weight of an edge exceeds this value then the subtasks that share this edge are declared heavily dependent and they are selected for static scheduling. The pseudo-code for implementing this heuristic is outlined in algorithm 7. Figure 5.4 shows the result of applying heuristic H3 on the TFG of Figure4.3.

Algorithm 7 Pseudo-code for Translation of Selection Heuristic H3

```
1:  $T \leftarrow \emptyset$;
2:  $Y \leftarrow \emptyset$;
3:  Evaluate $W_{th}$;
4:  **for** $e \in E^{TFG}$ **do**
5:      **if** EdgeWeight($e$) $> W_{th}$ **then**
6:          $T \leftarrow T \cup$ head($e$) $\cup$ tail($e$);
7:      **else**
8:          $Y \leftarrow Y \cup$ head($e$) $\cup$ tail($e$);
9:      **end if**
10: **end for**
```

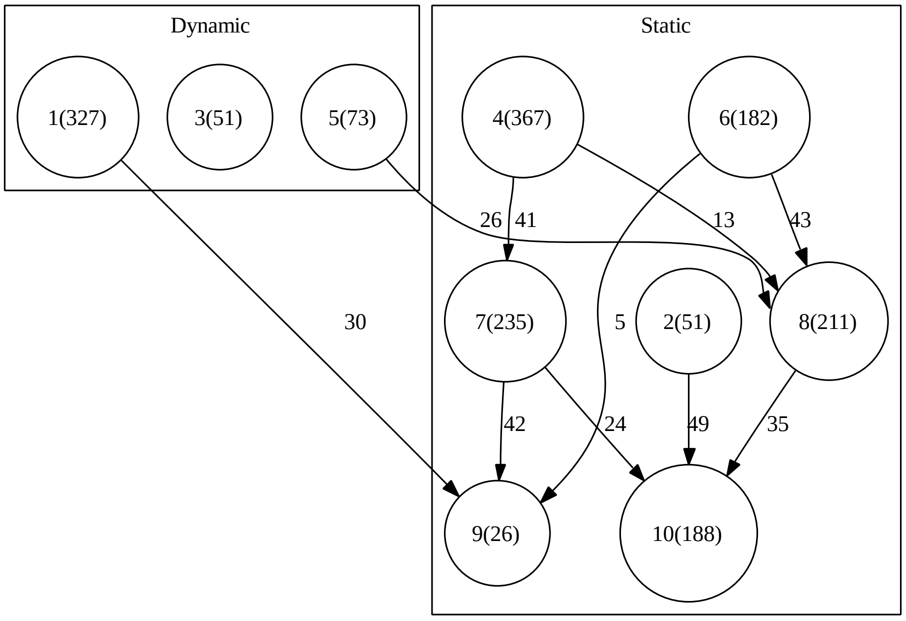


Figure 5.4. The TFG of Figure 4.3 after Application of Selection Heuristic H3

5.3.4 Selection Heuristic H4

The fourth heuristic, H4, identifies the nodes that can execute in parallel. It is achieved by *levelizing* the TFG: The root node(s) are at level 0, all the nodes that can be reached from root are at level 1, and so on. All the nodes in a level are data parallel nodes. They can be executed concurrently. If the number of such data parallel nodes exceeds a certain threshold, then the tasks in that level are scheduled dynamically. Formally:

$$H4 \rightarrow \begin{cases} Y = Y \cup \text{Nodes}(l) & \forall_{l \in \text{ levels in the TFG}} \text{If NumberOfNodes}(l) \geq P_{th} \\ Y = S - Y \end{cases}$$

The value P_{th} represents the average number of data-parallel nodes. It is currently evaluated as

$$P_{th} = \frac{|S|}{\text{NumberofLevels}(TFG)}$$

The pseudo-code for translating this heuristic is outlined in algorithm 8. Figure 5.5 depicts the effect of applying heuristic H4 to the input TFG of Figure 4.3. The function *Levelize(TFG)* of step 3 of algorithm 8 returns a list of nodes that are in the same level. The function *Nodes(l)* in step 7 of the algorithm returns a list of nodes in a given level l.

Algorithm 8 Pseudo-code for Translation of Selection Heuristic H4

```
1:  $T \leftarrow \emptyset$;
2:  $Y \leftarrow \emptyset$;
3:  Levels $\leftarrow$ Levelize($TFG$);
4:  Evaluate $P_{th}$;
5:  for $l$ in Levels do
6:      if Length($l$) $> P_{th}$ then
7:          $Y \leftarrow Y \cup$ Nodes($l$);
8:      end if
9:  end for
10: $T \leftarrow S|Y$;
```

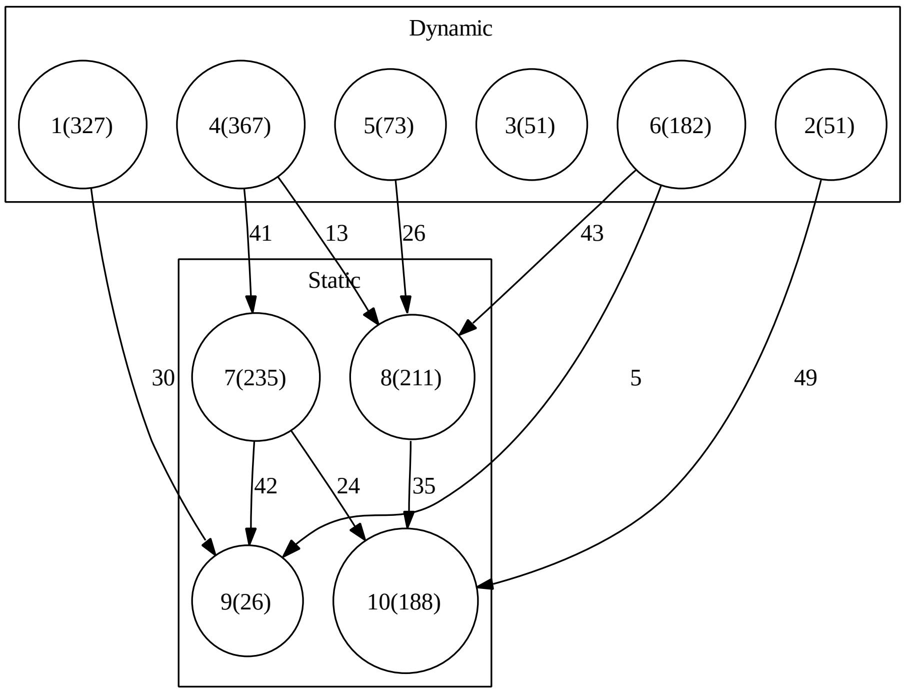


Figure 5.5. The TFG of Figure 4.3 after Application of Selection Heuristic H4

5.3.5 Selection Heuristic H5 and H6

Heuristics H5 and H6 use graph partitioning to translate the mix. H5 divides the TFG using standard bi-partitioning techniques with the criteria of partition being the edge cost. This results in two sub-graphs of TFG with minimum dependency between them. T and Y are determined from the sub-graphs. The partition criteria may also be changed to balancing the weights of the partition. In the latter case, both the schedulers have to schedule subgraphs with nearly equal computational requirements. Algorithm 9 shows the pseudo-code for implementing this heuristic. This heuristic is named H6. The results of applying these heuristics to the TFG of Figure 4.3 is shown in Figures 5.6 and 5.7. Figure 5.6 is obtained by using node cost as partition criteria and Figure 5.7 is obtained by using edge cost as the partitioning criteria. The partitioning of tasks was performed using METIS partitioning software.

Algorithm 9 Pseudo-code for Translation of Selection Heuristics H5 and H6

```
1: $T \leftarrow \emptyset$;
2: $Y \leftarrow \emptyset$;
3: $(T, Y) \leftarrow$ Partition($S$,Criteria);
```

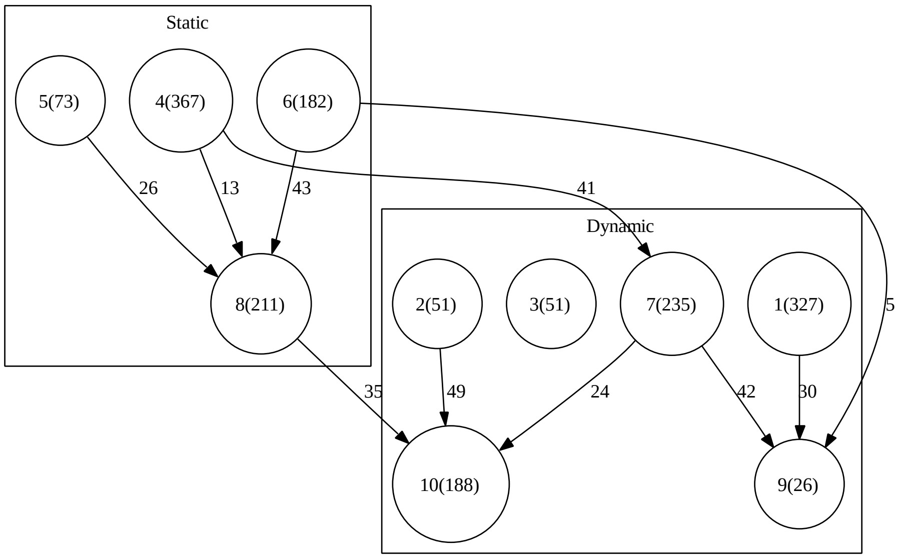


Figure 5.6. The TFG of Figure 4.3 after Application of Selection Heuristic H5 with criteria of partition being node weight

5.4 Heuristic based Dynamic Scheduling Algorithm

The heuristic scheduler differs from the dynamic scheduler in the fact that the heuristic scheduler has one additional constraint that must be satisfied by the machines in order to qualify as potential candidates. A machine is declared a candidate if the subtask can begin its execution on the machine no sooner than its slowest parent can finish and complete its execution in that machine and perform data transfer to all its dependents not later than its first dependent subtask can begin its execution. This is achieved by adjusting the Window(s_i) function. This additional constraint ensures that the static schedule is preserved. Algorithm 10 presents the pseudo-code for the proposed heuristic based dynamic scheduler. Algorithm 11 is modified to include the additional constraint. By comparing Figure 5.8 with the Figure 4.6 we can clearly see the advantage of this approach. The

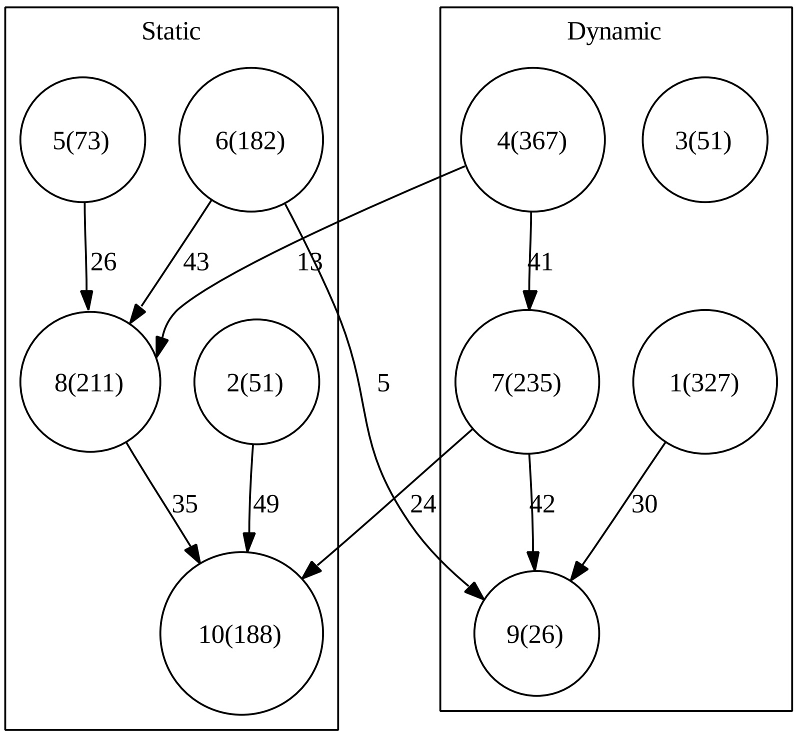


Figure 5.7. The TFG of Figure 4.3 after Application of Selection Heuristic H6 with criteria of partition being edge weight

number of buyers in the auction has decreased from 3 to 2. Also, it can be seen that the number of moves for each player has also been reduced.

5.5 Simulation Results and Discussion

Simulations were performed to study the effect of mix on solution quality and the to study effectiveness of the heuristics on the solution quality of the dynamic mapping. For the first set of experiments the data from Table 5.2 were used to generate a random task flow graph and a processor graph. Graphs with number of edges equal to $\frac{|S|}{3}$ are classified as applications with low data dependency. Those with ratios of $\frac{2|S|}{3}$ and $|S|$ are classified as graphs with medium and high data dependency [2]. From Figures 5.9, 5.10 and 5.11 we can infer that the completion times start to improve when the mix value is in the range of 0.4 to 0.3. The solution quality is best when $\mu = 0.1$, that is when 10% of tasks were

Algorithm 10 Heuristic Based Dynamic Scheduling Algorithm

```
1: Fix the value $\mu$
2: T,Y = TranslateMix (S,heuristics H1-H6)
3: StaticVSSA(S)
4: Levels = Levelize(S)
5: for List $\in$ tasks in each Level do
6:     for t $\in$ List do
7:         if t $in$ T then
8:             Append t to StaticList
9:         else
10:            Append t to DynamicList
11:        end if
12:    end for
13:    DynamicSchedule(DynamicList)
14: end for
```

Algorithm 11 Modified Window(s_i)

```
1: Set Window[$s_i$].LowerBound $\leftarrow$ Completion Time of the Slowest Parent
2: Set Window[$s_i$].UpperBound $\leftarrow$ Start Time of the First Dependent
3: Return Window
```

statically scheduled and 90% of the tasks were dynamically scheduled. The reason for this is due to the fact that the dynamic scheduler was able to use the information about the behavior of the application such as data transfer patterns from the static scheduler to improve the completion times of 90% of the tasks. It can also be inferred from the graph that for $\mu > 0.5$ there is no improvement in static schedule. To ensure the balanced use of both the static and dynamic schedulers, μ value of 0.3 is chosen

Table 5.1. Data used for Evaluating μ-parameter

Number of Tasks	10
Number of Machines	10
Number of Edges	3,6 and 10
Execution Matrix Data range	1000
Communication Matrix Data range	4
Data Exchange Matrix Data range	500

The second set of experiments is performed to compare the effectiveness of the six heuristics. For conducting this experiment, a library of random task graphs and processor

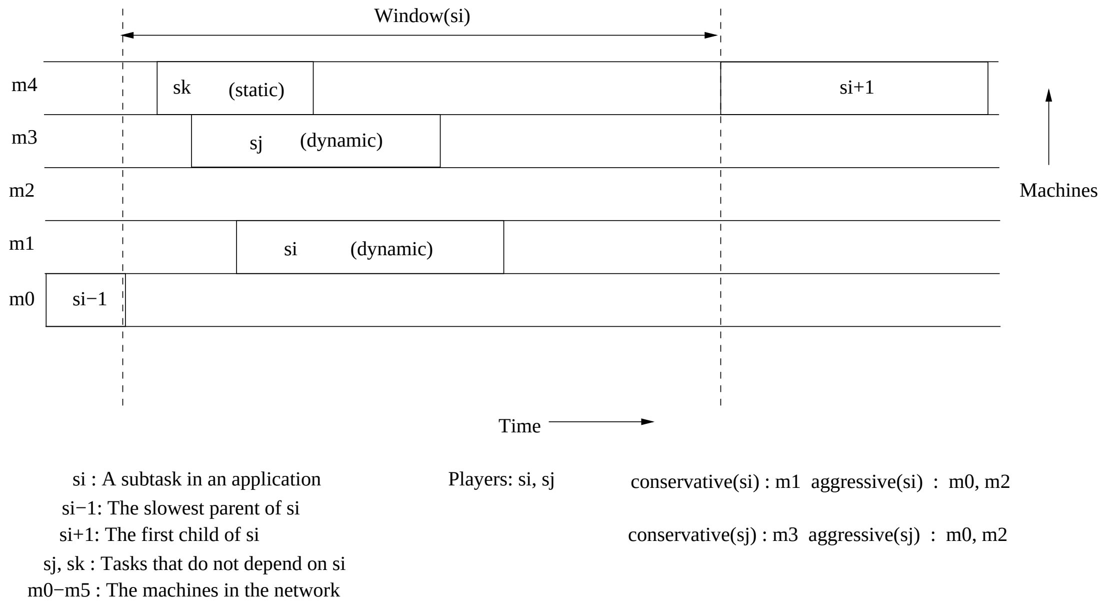

Window(si)
m4
sk (static)
si+1
m3
sj (dynamic)
m2
Machines
m1
si (dynamic)
m0
si−1
Time
si : A subtask in an application
si−1: The slowest parent of si
si+1: The first child of si
sj, sk : Tasks that do not depend on si
m0−m5 : The machines in the network
Players: si, sj
conservative(si) : m1 aggressive(si) : m0, m2
conservative(sj) : m3 aggressive(sj) : m0, m2

Figure 5.8. Freeze-frame Illustrating the System State when s_i, s_j and s_k are being Dynamically Scheduled using the Proposed Heuristics on the Game Theoretic Scheduler

graphs were created using data from Table 5.2. Random task flow graphs were generated with node sizes 10, 50 and 100 for low, medium and high data dependencies. Random processor graphs were generated for node sizes of 5, 10 and 20 processors.

From the graphs in Figures 5.12 - 5.20 it is observed that heuristic H2 provides better results for most of the situations. It is also observed that for TFG with large sizes, heuristics H5 and H6 also provide better results.

It is to be noted that the proposed heuristics are used for the random graphs that are generated. However, it would be worthwhile to investigate heuristics, other the proposed ones, that exploit a specific feature of the application. In essence, μ and the heuristics are parameters that are highly dependent on the application that is being mapped to the HC suite.

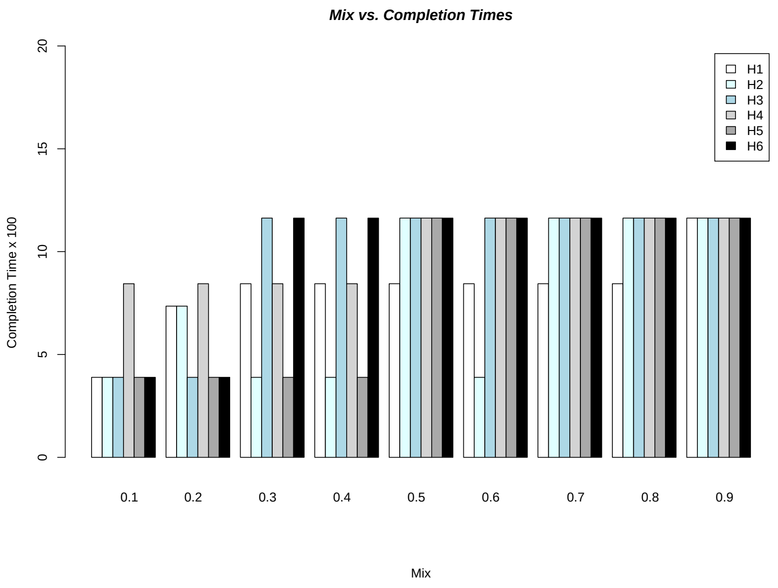


Figure 5.9. Mix v. Completion Time for $|M| = 10$, $|S| = 10$ with Low Dependency

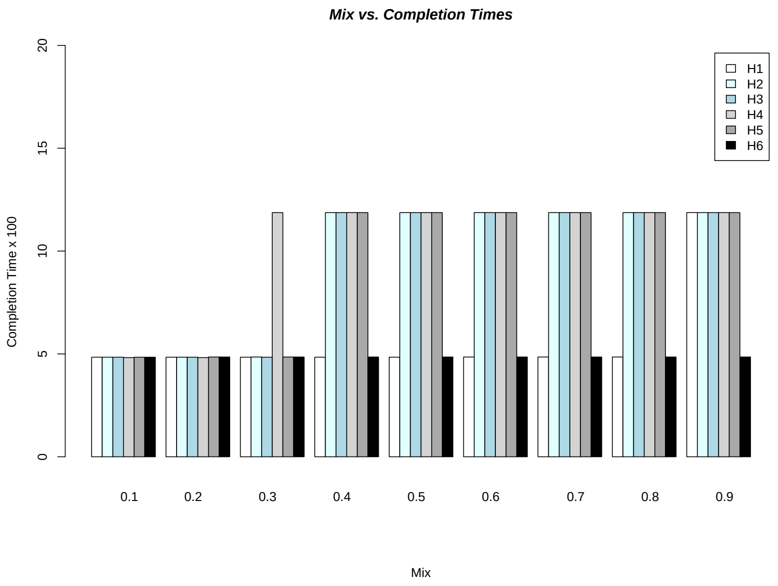


Figure 5.10. Mix v. Completion Time for $|M| = 10$, $|S| = 10$ with Medium Dependency

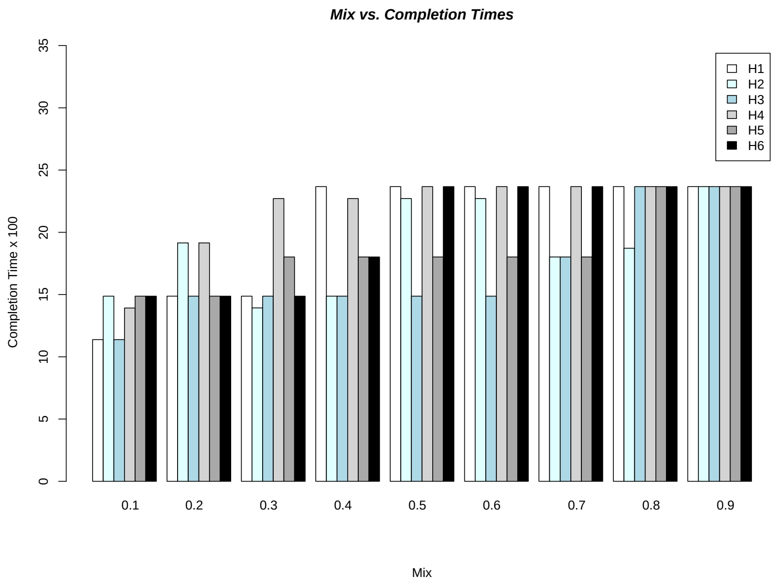


Figure 5.11. Mix v. Completion Time for $|M| = 10$, $|S| = 10$ with High Dependency

Table 5.2. Data used for Evaluating Heuristics

Number of Tasks $\vert S\vert$	10, 50, 100
Number of Machines $\vert M\vert$	5,10,20
Number of Edges	$\frac{\vert S\vert}{3}$, $\frac{2\vert S\vert}{3}$,$\vert S\vert$
Execution Matrix Data range	1000
Communication Matrix Data range	4
Data Exchange Matrix Data range	500

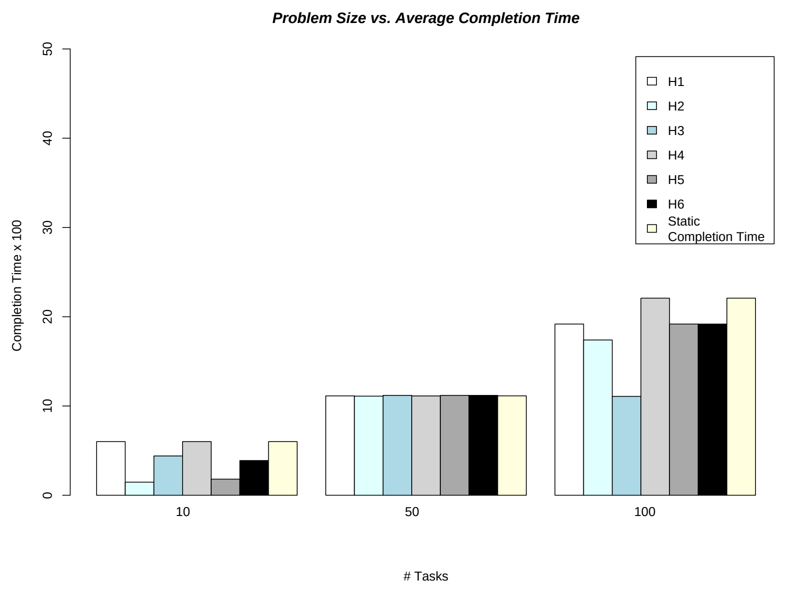


Figure 5.12. Size vs. Completion Time for $|M| = 5$ and with Low Data Dependency

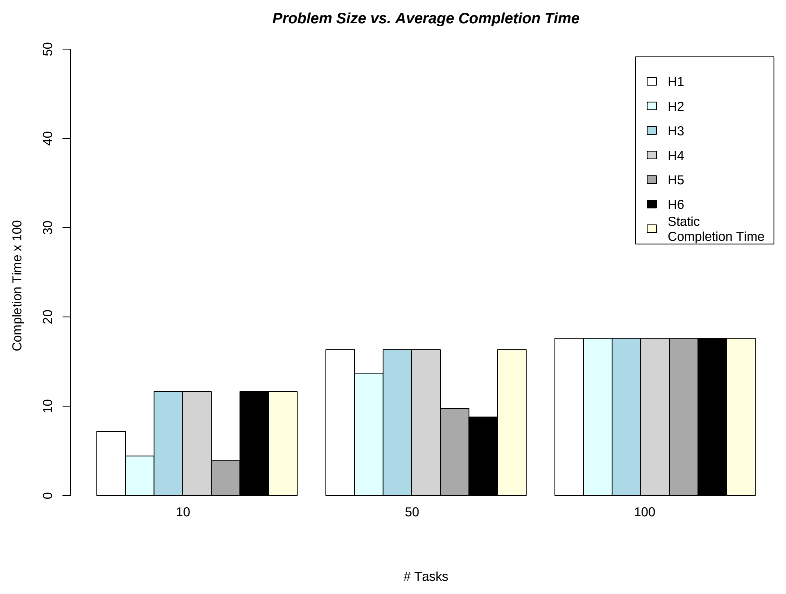


Figure 5.13. Size vs. Completion Time for $|M| = 10$ and with Low Data Dependency

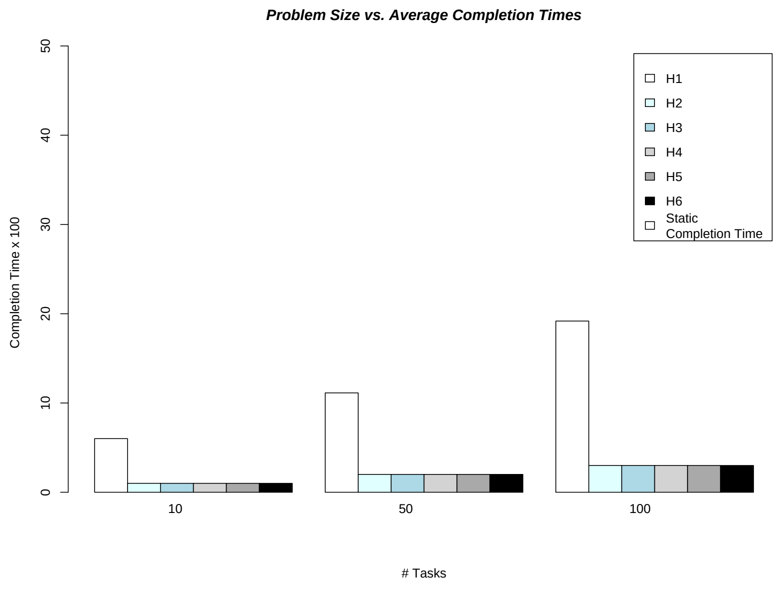


Figure 5.14. Size vs. Completion Time for $|M| = 20$ and with Low Data Dependency

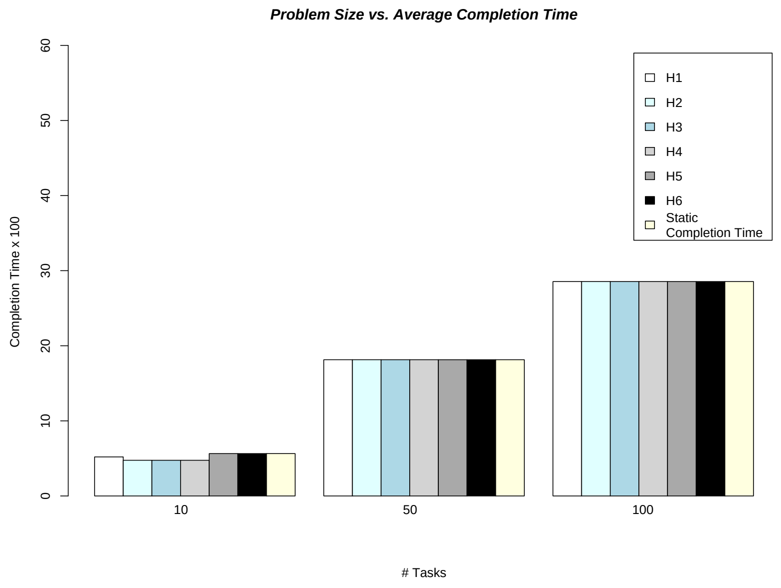


Figure 5.15. Size vs. Completion Time for $|M| = 5$ and with Medium Data Dependency

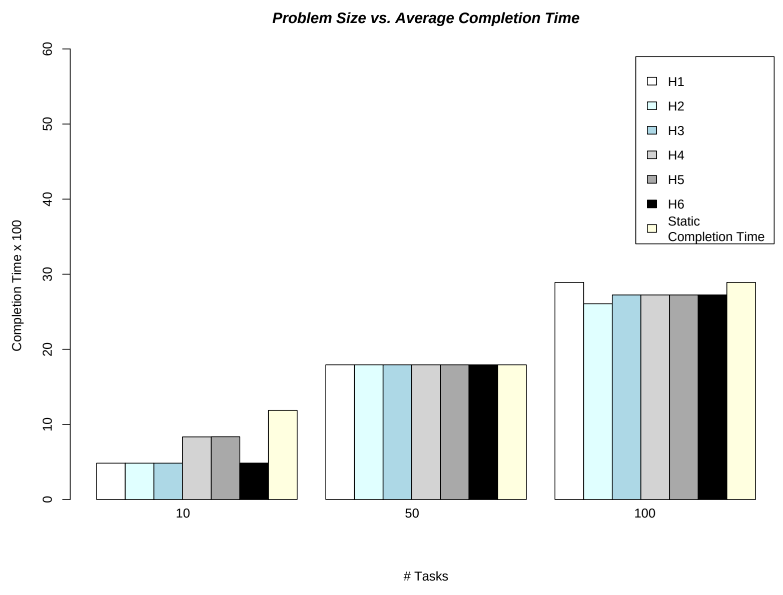


Figure 5.16. Size vs. Completion Time for $|M| = 10$ and with Medium Data Dependency

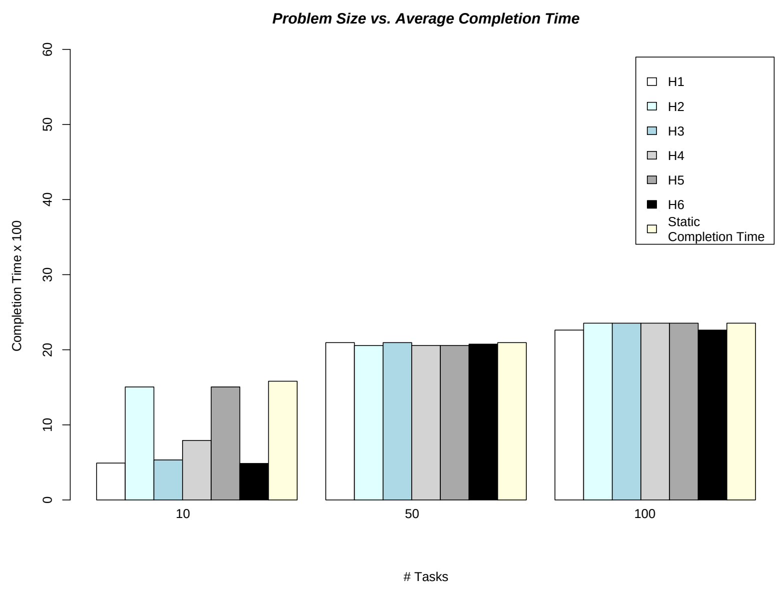


Figure 5.17. Size vs. Completion Time for $|M| = 20$ and with Medium Data Dependency

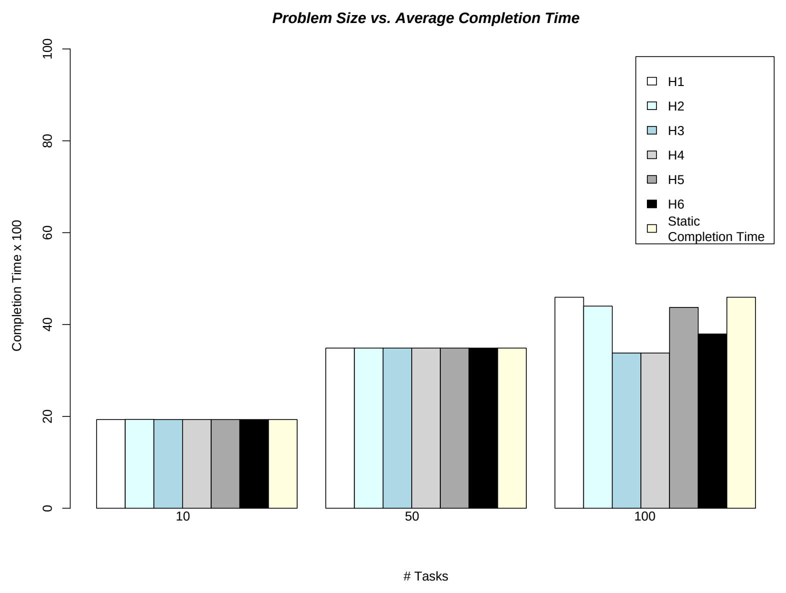


Figure 5.18. Size vs. Completion Time for $|M| = 5$ and with High Data Dependency

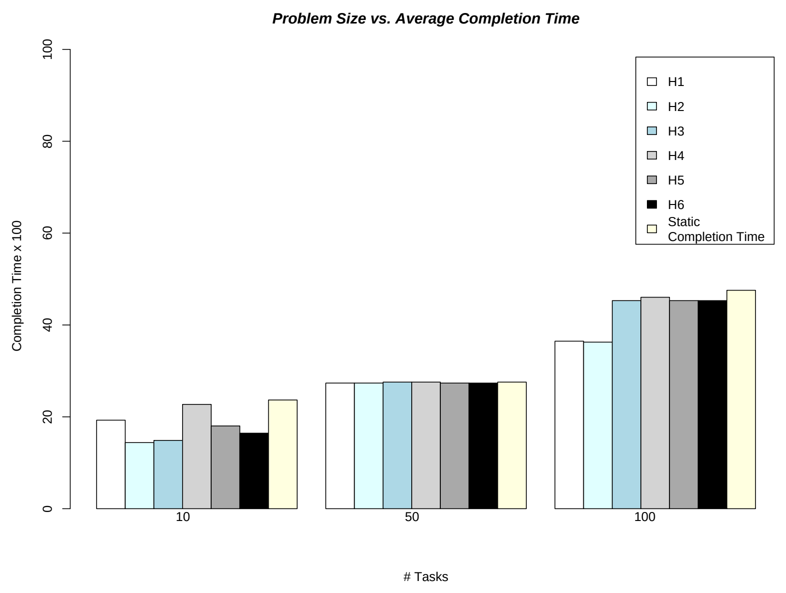


Figure 5.19. Size vs. Completion Time for $|M| = 10$ and with High Data Dependency

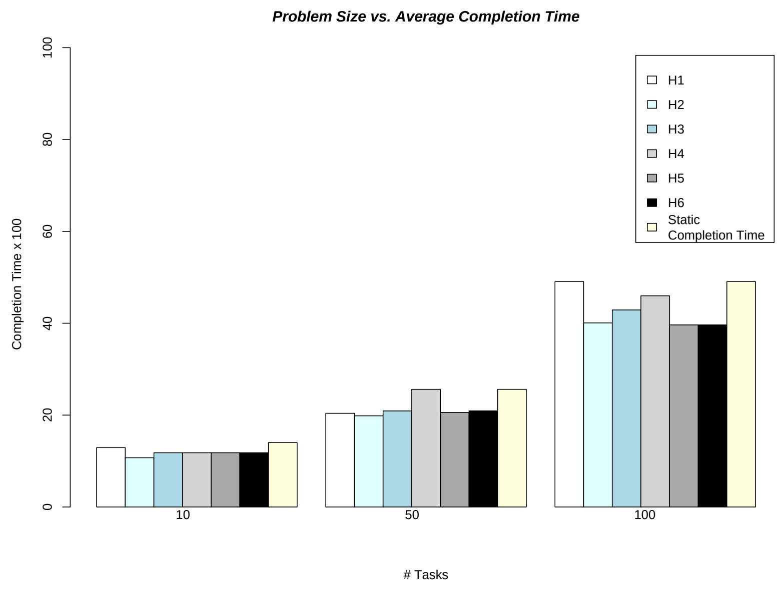


Figure 5.20. Size vs. Completion Time for $|M| = 20$ and with High Data Dependency

CHAPTER 6

CONCLUSIONS

Heterogeneous computing provides a structured methodology to exploit the diversity in the application and computational domain. There are two main reasons why HC is important for the construction of high performance computing systems: (i) most HPC systems achieve only a fraction of their peak performance on real application sets, and (ii) the economic viability of HC systems. Among the issues in developing HC systems, task assignment and scheduling is identified to be the most critical. The contributions of the thesis are:

- A new scheduling framework based auctions based on game theory.
- Task assignment algorithms with six heuristics to reduce the scheduling overhead.

  The techniques presented in the thesis are a first step in this direction. It is possible to enhance the speed of execution of the proposed algorithms, by studying new heuristics and using creative techniques to solve the Nash equilibrium. It will be worthwhile to study the performance of the proposed techniques for specific high performance applications.

REFERENCES

[1] R. F. Freund, “SuperC or Distributed Heterogenous HPC,” in *Computing Systems in Engineering*, 1991, vol. 2, pp. 349–355.

[2] Raju.D.Venkataramana, *Task Assignment And Scheduling Algorithms For Heterogenous Computing Systems*, Ph.D. thesis, University of South Florida, August 2000.

[3] Muhammad Kafil and Ishfaq Ahmad, “Optimal Task Assignment in Heterogenous Computing Systems,” in *Proceedings of Sixth Heterogenous Computing Workshop (HCW’97)*, April 1997, pp. 135–146.

[4] H. Topcuoglu, S. Hariri, and M. Y. Wu, “Task scheduling algorithms for heterogenous processors,” in *Proceedings of Heterogenous Computing Workshop*, 1999.

[5] R.F.Freund and H.J.Seigel, “Heterogenous processing,” in *IEEE Computer*, 1993, vol. 26, pp. 88–95.

[6] A.A.Khokhar, V.K.Prasanna, M.E.Shaaban, and C.Wang, “Heterogenous computing: Challenges and opportunities,” in *IEEE Computer*, 1993, vol. 26, pp. 18–27.

[7] H.J.Seigel et al., “Report of the purdue workshop on grand challenges in computer architecture for the support of high-performance computing,” in *Journal of Parallel and Distributed Computing*, 1992, vol. 16, pp. 199–211.

[8] H.J.Seigel et al., *Parallel Computing: Paradigms and Applications*, chapter 25, pp. 78–114, International Thomson Computer Press, London, 1996.

[9] C.C.Weems et al., “The image understanding architecture,” in *International Journal of Computer Vision*, 1989, vol. 2, pp. 251–282.

[10] V.S. Sunderam, “PVM: a framework for parallel distributed computing,” in *Concurrency: Practise and Experience*, 1990, vol. 2, pp. 315–339.

[11] Message Passing Interface Forum, “MPI: a message-passing interface standard,” in *International Journal of Supercomputer Applications*, 1994.

[12] N.Carriero, D. Gelernter, and T.G.Mattson, “Linda in heterogenous computing environments,” in *Proceedings of Workshop on Heterogenous Processing*, 1992, pp. 43–46.

[13] R.M. Butler and E.L. Lusk, “Monitors,messages and clusters: The p4 parallel programming system,” in *Parallel Computing*, 1994, vol. 20, pp. 547–564.

[14] A.S.Grimshaw, "Easy-to-use object-oriented parallel processing with mentat," in *IEEE Computer*, 1993, vol. 26, pp. 39–51.

[15] A.L.Beguelin, J. Dongarra, G.A. Geist, R. Manchek, and V.S. Sunderam, "Visualization and debugging in a heterogenous environment," in *IEEE Computer*.

[16] A.Y.H. Zomaya, *Parallel and Distributed Computed Handbook*, Mc Graw-Hill Publishers, New York, NY, 1996.

[17] R. F. Freund, "Optimal Selection Theory for Superconcurrency," in *Proceedings Supercomputing '89*, November 1989, pp. 699–703.

[18] E. Arnould et al., "The design of nectar: A network backplane for heterogenous multicomputers," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, IEEE-CS Press, Los Alamitos, California, 1989, pp. 205–216.

[19] ANSI X3T9.3, "High-performance parallel interface: Hippi-ph, hippi-sc, hippi-fp, hippi-le, hippi-mi," in *American National Standard for Information Systems*, American National Standards Institute,New York, 1991.

[20] D. Arapov, A. Kalinov, A. Lastovetsky, and I Ledovskih, "A programming environment for heterogenous distributed memory machines," in *Proceedings of Heterogenous Computing Workshop (HCW'97)*, April 1997.

[21] H.S.Stone, "Multiprocessor Scheduling with the aid of Network Flow Algorithms," January 1977, vol. 3, pp. 85–93.

[22] V.M.Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," 1988, vol. 37, pp. 1384–1397.

[23] M.M.Eshagian and Y.C.Wu, "Mapping Heterogenous Task Graphs onto Heterogenous System Graphs," in *Proceedings of Sixth Heterogenous Computing Workshop (HCW '97)*, April 1997, pp. 147–160.

[24] M. Tan et al., "Scheduling and Data Relocation for Sequentially Executed Subtasks in a Heterogenous System," in *Proceedings of Fourth Heterogenous Computing Workshop (HCW'95)*, 1995, pp. 109–120.

[25] A. V. Aho, J.E.Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, MA, 1974.

[26] C. Leangsuksun and J. Potter, "Designs and Experiments on Heterogenous Mapping Heuristics," in *Proceedings of Third Heterogenous Computing Workshop (HCW'94)*, 1994, pp. 17–22.

[27] R. Armstrong, D. Hensgen, and T. Kidd, "The relative performance of various mapping algorithms is independent of sizeable variances in run-time predictions," in 7^{th} *IEEE Heterogenous Computing Workshop (HCW'98)*, March 1998, pp. 79–87.

[28] R. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. Lima, F. Mirabile, L. Moore, B. Rust, and H. Seigel, “Scheduling resources in multi-user heterogenous computing environments using SmartNet,” in 7^{th} *IEEE Heterogenous Computing Workshop (HCW’98)*, March 1998, pp. 184–199.

[29] O. Ibarra and C. Kim, “Heuristic algorithms for scheduling independent tasks on nonidentical processors,” in *Journal of the ACM*, April 1977, pp. 280–289.

[30] M. Y. Wu, W. Shu, and H. Zhang, “Segmented min min: A static mapping algorithm fr meta-tasks on heterogenous computing systems,” in *Proceedings of Heterogenous Computing Workshop*.

[31] M. Wang et al., “Augmenting the Optimal Selection Theory for Superconcurrency,” in *Proceedings of the Workshop on Heterogenous Processing*, 1992, pp. 13–22.

[32] S. Chen et al., “Selection Theory for Methodology for Heterogenous Supercomputing,” in *Proceedings of Heterogenous Computing Workshop (HCW’93)*, 1993, pp. 15–22.

[33] B. Narahari, A. Youssef, and H.A.Choi, “Matching and Scheduling in a Generalized Optimal Selection Theory,” in *Proceedings of Heterogenous Computing Workshop(HCW’94)*, 1994, pp. 3–8.

[34] M. Coli and P. Palazzari, “Real time piplined system design through simulated annealing,” December 1996, vol. 42, pp. 465–475.

[35] S. Krikpatrick, C. D. Gelatt Jr., and M. P. Vecchi, “Optimization by simulated annealing,” in *Science*, May 1983, vol. 220, pp. 671–680.

[36] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[37] I. De Falco, R. Del Balio, E. Tarantino, and R. Vaccaro, “Improving search by incorporating evolution principles in parallel tabu search,” in 1994 *IEEE Conference on Evolutionary Computation*, 1994, vol. 2, pp. 823–828.

[38] F. Glover and M. Laguna, *Tabu Search*, Kluwer Academic Publishers, Boston, MA, June 1997.

[39] H. Singh and A. Youssef, “Mapping and scheduling heterogenous task graphs using genetic algorithms,” in 5^{th} *IEEE Heterogenous Computing Workshop (HCW’96)*, April 1996, pp. 86–97.

[40] L. Wang, H. J. Seigel, V. P. Roychowdhury, and A. A. Maciejewski, “Task matching and scheduling in heterogenous computing environments using a genetic-algorithm-based approach,” in *Journal of Parallel and Distributed Computing*, November 1997.

[41] J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

[42] D.Whitley, "The Genitor Algorithm and Selection Pressure:why Rank-based allocation of reproductive Trials is the Best," in *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[43] A. Brindle, "GA for Function Optimization," Tech. Rep., 1981.

[44] J.Baker, "Reducing Bias and Inefficiency in the Selection Algorithm," in *Proceedings of the Second International conference on Genetic Algorithms*, 1989.

[45] D. Goldberg, *Genetic Algorithms in Search, optimization and Machine Learning*, Addison-Wesley, MA, 1989.

[46] A. Mahmood, "A hybrid genetic algorithm for task scheduling in multiprocessor real-time systems," in *Studies in Informatics and Control Journal*, 2000, vol. 9, pp. 207–218.

[47] T. D. Braun, H. J. Seigel, N. Beck, L. L. B$\ddot{o}$l$\ddot{o}$ni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison study of static mapping heuristics for a class of meta-tasks on heterogenous computing systems," in *Proceedings of Heterogenous Computing Workshop*.

[48] N. S. Flann H. Chen and D. W. Watson, "A massively parallel simd approach," in *IEEE Transactions on Parallel and Distributed Computing*, February 1998, vol. 9, pp. 126–136.

[49] P. Shroff, D. Watson, N. Flann, and R. Freund, "Genetic simulated annealing for scheduling data dependent tasks in heterogenous environments," .

[50] R. D. Venkataramana and N. Ranganathan, "New cost metrics for iterative task assignment algorithms in heterogenous computing systems," in *Proceedings of Heterogenous Computing Workshop*, May 2000.

[51] R .D. Venkataramana and N. Ranganathan, "Multiple Cost Optimization for Task Assignment in Heterogenous Computing Systems Using Learning Automata," in *Proceedings of Heterogenous Computing Workshop*, April 1999, pp. 137–145.

[52] M. Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," in *IEEE Transactions on Parallel and Distributed Systems*, July 1990, pp. 330–343.

[53] G. L. Park, B. Shirazi, J. Marquis, and H. Choo, "Decisive path scheduling: A new list scheduling method," in *Proceedings of International Conference on Parallel Processing*, 1997.

[54] A. Radulescu and A. J. C. van Gemund, "On the complexity of list scheduling algorithms for distributed-memory systems," in *Proceedings of ACM International Conference on Supercomputing*, 1999.

[55] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," in *SIAM Journal on Computing*, April 1989, pp. 244–257.

[56] C. Y. Lee, J. J. Hwang, Y. C. Chow, and F. D. Anger, "Multiprocessor scheduling with interprocessor communication delays," in *Operations Research Letters*, June 1988, pp. 141–147.

[57] A. Radulescu and A. J. C. van Gemund, "Flb: Fast load balancing for distributed-memory machines," in *Proceedings of International Conference on Parallel Processing*, 1999.

[58] G. C. Sih and E. A. Lee, "A compile time scheduling heuristic for interconnection-constrained heterogenous processor architectures," in *IEEE Transactions on Parallel and Distributed Systems*, February 1993, vol. 4, pp. 175–186.

[59] M. Iverson, F. Ozguner, and G. Follen, "Parallelizing existing applications in a distributed heterogenous environments," in *Proceedings of Heterogenous Computing Workshop*, 1995, pp. 93–100.

[60] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," in *Journal of Parallel and Distributed Computing*, 1990, vol. 9, pp. 138–153.

[61] M. Maheswaran and H. J. Seigel, "A dynamic matching and scheduling algorithm for heterogenous computing systems," in *Proceedings of Heterogenous Computing Workshop*, 1998.

[62] John Von Neumann, "Zur theories der gesellschaftsspiele," in *Mathematische Annalen*, 1928, pp. 100:295–320.

[63] J. Nash, "Noncooperative games," in *Annals of Mathematics*, 1951, pp. 54:289–295.

[64] Anatol Rapoport, *N-Person Game Theoy Concepts and Applications*, The University of Michigan Press.

[65] J. Nash, "Equilibrium points in n-person games," in *Proceedings of The National Academy of Sciences U. S. A.*, 1951, pp. 36:48–49.

[66] Richard Mckelvey, Andrew McLennan, and Theodore Turocy, *Gambit: Software Tools for Game Theory*, Texas A & M University.

[67] A. Y. Zomaya, "Observations on using genetic algorithms for dynamic load-balancing," in *IEEE Transactions on Parallel and Distributed Systems*, 2001, vol. 12.

[68] Andrew J. Page and Thomas J. Naughton, "Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing," in *Proceedings of the 19th IEEE/ACM International Parallel and Distributed Processing Symposium*, Denver, Colorado, USA, April 2005, IEEE Computer Society.

[69] M. D. Theys, T. D. Braun, H. J. Siegal, A. A. Maciejewski, and Y.-K. Kwok, *Mapping Tasks onto Distributed Heterogeneous Computing Systems Using a Genetic Algorithm Approach*, John Wiley and Sons, New York, USA.

[70] R. D. Venkataramana and N. Ranganathan, "A Learning Automata Based Framework for Task Assignment in Heterogenous Computing Systems," in *ACM Symposium on Applied Computing*, 1999.

[71] R. Freund and H. J. Seigel, "Heterogenous processing," in *IEEE Computer*, June 1993, pp. 13–17.

[72] K. Taura and A. Chien, "A heuristic algorithm for mapping communicating tasks on heterogenous resources," in *Proceedings of Heterogenous Computing Workshop*.

[73] M. Maheswaran, S. Ali, H. J. Seigel, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogenous computing systems," in *Proceedings of Heterogenous Computing Workshop*.