ALGORITHMS FOR TASK SCHEDULING IN HETEROGENEOUS

COMPUTING ENVIRONMENTS

Except where reference is made to the work of others, the work described in this dissertation is my own or was done in collaboration with my advisory committee. This dissertation does not include proprietary or classified information.

_____

Prashanth C. Sai Ranga

Certificate of Approval:

_____	_____
Homer W. Carlisle Associate Professor Computer Science and Software Engineering	Sanjeev Baskiyar, Chair Associate Professor Computer Science and Software Engineering
_____	_____
Yu Wang Assistant Professor Computer Science and Software Engineering	Joe F. Pittman Interim Dean Graduate School

ALGORITHMS FOR TASK SCHEDULING IN HETEROGENEOUS COMPUTING ENVIRONMENTS

Prashanth C. Sai Ranga

A Dissertation

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Doctor of Philosophy

Auburn, Alabama
December 15, 2006

ALGORITHMS FOR TASK SCHEDULING IN HETEROGENEOUS

COMPUTING ENVIRONMENTS

Prashanth C. Sai Ranga

_____

Signature of Author

_____

Date of Graduation

DISSERTATION ABSTRACT

ALGORITHMS FOR TASK SCHEDULING IN HETEROGENEOUS COMPUTING ENVIRONMENTS

Prashanth C. Sai Ranga

Doctor of Philosophy, Dec 15,2006
(M.S., University of Texas at Dallas, Dec, 2001)
(B.E., Bangalore University, India, Aug 1998)

136 Typed pages

Directed by Sanjeev Baskiyar

Current heterogeneous meta-computing systems, such as computational clusters and grids offer a low cost alternative to supercomputers. In addition they are highly scalable and flexible. They consist of a host of diverse computational devices which collaborate via a high speed network and may execute high-performance applications. Many high-performance applications are an aggregate of modules. Efficient scheduling of such applications on meta-computing systems is critical to meeting deadlines. In this dissertation, we introduce three new algorithms, the Heterogeneous Critical Node First (HCNF) algorithm, the Heterogeneous Largest Task First (HLTF) algorithm and the Earliest Finish Time with Dispatch Time (EFT-DT) algorithm. HCNF is used to schedule

parallel applications of forms represented by directed acyclic graphs onto networks of workstations to minimize their finish times. We compared the performance of HCNF with those of the Heterogeneous Earliest Finish Time (HEFT) and Scalable Task Duplication based Scheduling (STDS) algorithms. In terms of Schedule Length Ratio (SLR) and speedup, HCNF outperformed HEFT on average by 13% and 18% respectively. HCNF outperformed STDS in terms of SLR and speedup on an average by 8% and 12% respectively. The HLTF algorithm is used to schedule a set of independent tasks onto a network of heterogeneous processors to minimize finish time. We compared the performance of HLTF with that of the Sufferage algorithm. In terms of makespan, HLTF outperformed Sufferage on average by 4.5 %, with a tenth run-time. The EFT-DT algorithm schedules a set of independent tasks onto a network of heterogeneous processors to minimize finish time when considering dispatch times of tasks. We compared the performance of EFT-DT with that of a First in First out (FIFO) schedule. In terms of minimizing makespan, on average EFT-DT outperformed FIFO by 30%.

ACKNOWLEDGMENTS

The author is highly indebted to his advisor, Dr. Sanjeev Baskiyar, for his clear vision, encouragement, persistent guidance and stimulating technical inputs. His patience, understanding and support are deeply appreciated. Thanks to Dr. Homer Carlisle and Dr. Yu Wang, for their review and comments on this research work. Their invaluable time spent on serving on my graduate committee is sincerely appreciated. Special thanks to Mr. Victor Beibighauser, Mr. Basil Manly and Mr. Ron Moody of South University, Montgomery, for their concern, understanding and co-operation. Finally, the author would like to thank his parents, sister and bother-in-law for their constant support and encouragement.

Style manual or journal used: IEEE Transactions on Parallel and Distributed Systems

Computer software used: Microsoft Word, Adobe PDF

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

This chapter provides an introduction to our research work and discusses a few relevant topics. Section 1.1 discusses our research motivation. Section 1.2 describes the architecture of cluster computing systems. Section 1.3 describes the architecture of grid computing systems. Section 1.4 provides an overview of task scheduling in heterogeneous computing systems. Section 1.5 provides an introduction to NP-complete problems and Section 1.6 discusses the organization of this dissertation.

1.1 Motivation

Information Technology has revolutionized the way we share and use information. The IT revolution has witnessed a myriad number of applications with a wide range of objectives which include: small personal computer based applications like the calculator program, medium-sized applications like the Microsoft Word, large-sized applications like the Computer Aided Design software and very-large sized applications like the Weather Forecasting application. Some of these programs can run efficiently on a normal personal computer and some may need a more powerful workstation. However, there are applications like Weather Forecasting, Earthquake Analysis, Particle Simulation and a host of other engineering and scientific applications that require computing

capabilities beyond that of personal computers or workstations. They are called "High-Performance Applications".

How do we run these high-performance applications efficiently, given the fact that sequential computers (PCs, workstations) are too slow to handle them? There are three ways to improve efficiency [1]: work harder, work smarter or get help. In this context, working harder refers to increasing the speed of sequential uni-processor computers. In the last two decades, microprocessor speed has on an average doubled once in 18 months. Today's microprocessor chip is faster than the mainframes of yesteryears, owing to the phenomenal advances in Very Large Scale Integration (VLSI) technology. Even though this trend is expected to continue in the future, microprocessor speed is severely limited by the laws of physics and thermodynamics [2]. There is very high probability that it will eventually hit a plateau in the near future.

Working smarter refers to designing efficient algorithms and programming environments to deal with high-performance applications. By working smarter, we can definitely improve the overall efficiency, but will not be able to overcome the speed bottleneck of sequential computers.

Getting help refers to involving multiple processors to solve the problem. The idea of multiple processors working together simultaneously to run an application is called "Parallel Processing." Most of the applications consist of thousands of modules or sub-programs that may or may not interact with each other depending on the nature of the application. In either case, there are usually a number of modules that are independent of one another and could run simultaneously on different processors. The parallel nature of many applications is what makes parallel processing very appealing. In other words, if

applications were to be one large sequential module, parallel processing would not be feasible.

Parallel processing has captivated researchers for a long time. The initial trend in parallel processing was to create tightly coupled multi-processor systems with shared memory, running proprietary software. These systems were generally referred to as "Supercomputers". Supercomputers were extremely fast and expensive. In the 1960s Seymour Cray created the world's first commercial supercomputer the CDC 6600. Other companies like IBM, Digital and Texas Instruments created their own proprietary versions of supercomputers. The 70s and the 80s witnessed major companies and research labs across the word vie with one another to create the world's fastest super computer. Even though the trend continues to this day, parallel processing has slowly drifted away from supercomputing for a number of reasons. Supercomputers are extremely expensive systems that run on proprietary technology. Since they run on proprietary technology, they offer less flexibility with respect to developing software solutions to execute high performance applications. Since supercomputers are very expensive to lease/purchase and maintain, it is beyond the reach of many organizations to deploy them. Also in view of today's technological growth, it is important for systems to be readily scalable. Owing to factors like proprietary hardware and software technologies, most of the supercomputers are not readily scalable. To summarize, supercomputers have a very high cost/performance factor.

The very high cost/performance factor made them unattractive to a number of organizations. Most organizations (business, academic, military etc) were interested in high performance computing but were seeking systems with low cost/performance factor,

which could not be offered by supercomputers. In the meantime, PCs and workstations became extremely powerful and significant advances were made in networking technologies. Researchers began to explore the possibility of connecting low cost PCs with a high-speed network to mimic the functioning of a supercomputer albeit with a low cost/performance factor.

Extensive research has been carried out to create high performance systems by connecting PCs/workstations with a high-speed network. Most of the research was focused on creating viable parallel programming environments, developing high-speed network protocols and devising effective scheduling algorithms. Initially, the PCs/workstations had uniform hardware characteristics and thus the systems were termed "Homogeneous." However due to rapid advances in PC technology, computers and other hardware items had to be continuously upgraded and it was no longer the case that all the machines had identical hardware characteristics. This led to the notion of "Heterogeneous Systems" where individual PCs/workstations in a network could have different hardware characteristics. Researchers today focus on creating a high-performance system with a low cost/performance factor using a Heterogeneous Network of Workstations (NOWs).

So, what goes into creating a viable high performance computing system with a low cost/performance ratio out of a NOW given the fact that we have powerful workstations and very high-speed networks? Firstly, an efficient run-time environment must be provided for high-performance applications. Extensive research has been done in this area and has led to the creation of efficient technologies like the Message Passing Interface (MPI) [2] and the Parallel Virtual Machine (PVM) [2]. Secondly, in order to be able to provide a low cost/performance ratio, these systems must optimize the overall

execution time (or turnaround time) of high-performance applications. This requires efficient scheduling of the sub-tasks of high-performance applications onto the individual machines of a NOW. The sub-tasks of a parallel application may either be independent or may have precedence constraints. In either case, the problem of scheduling these subtasks to optimize the overall execution time of an application is a well-known NP Complete problem [3].

The focus of our research is to devise efficient scheduling algorithms for scheduling parallel applications represented by independent tasks as well as tasks with precedence constraints onto heterogeneous computing systems to minimize the overall execution time. We strongly believe that efficient task scheduling is the most important factor in creating a low-cost high-performance computing system. We now discuss the architectures of two very popular heterogeneous computing systems, the Cluster and the Grid.

1.2 Cluster Computing

A cluster is a heterogeneous parallel computing system which consists of several stand alone systems that are interconnected to function as an integrated computing resource. A cluster generally refers to two of more computers interconnected via a local area network. A cluster of computers can appear as a single system to users and applications. It provides a low-cost alternative to supercomputers with a relatively reasonable performance.

Figure 1.1 describes the architecture and the main components of a cluster computing system [2]. The individual nodes of a cluster could be PCs or high speed

workstations connected through a high-speed network. The network interface hardware acts as a communication processor and is responsible for transmitting and receiving packets of data between cluster nodes. The cluster communication software provides a means for fast and reliable data communication among cluster nodes and to the outside world. Clusters often use communication protocols such as "Active Messages" [2] for fast communication among their nodes. They usually bypass the operating system and remove the critical communication overhead normally involved by providing a direct user-level access to the network interface.

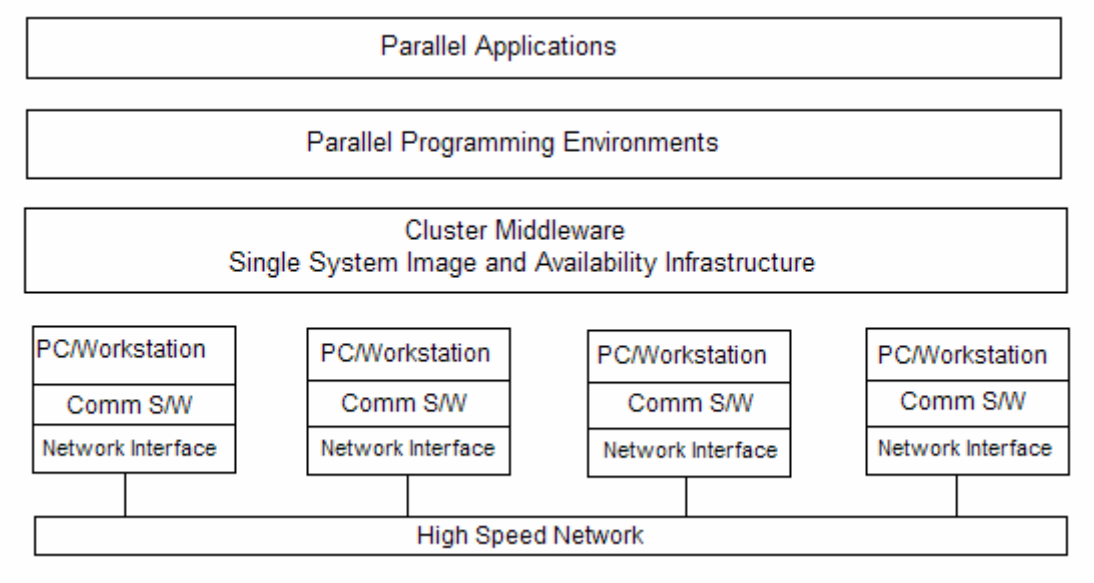

Figure 1.1 Architecture of Cluster-Computing Systems

The cluster nodes can either work as individual computers or can work collectively as an integrated computing resource. The cluster middleware is responsible for offering an illusion of a unified system image (Single System Image) and Availability

out of a collection of independent but interconnected computers. Parallel programming environments offer portable, efficient, and easy-to-use tools for development of applications. They include message passing libraries, debuggers, and profilers. Clusters also run resource management and scheduling software such as LSF (Load Sharing Facility) and CODINE (Computing in Distributed Networked Environments) [2]. The individual nodes of a cluster can have different hardware characteristics and new nodes can be seamlessly integrated into existing clusters thus making them easily scalable. Clusters make use of these hardware and software resources to execute high performance applications and typically provide a very low cost/performance ratio.

1.3 Grid Computing

The massive growth of the Internet in the recent years has encouraged many scientists to explore the possibility of harnessing idle CPU clock cycles and other unutilized computational resources spread across the Internet. The idea was to harness idle CPU cycles and other computational resources and provide a unified computational resource to those in need of high-performance computation. This led to the notion of "Grid Computing".

The concept of grid computing is similar to that of "Electrical Grids." In electrical grids, power generation stations in different geographical locations are integrated to provide a unified power resource for consumers to plug into on demand. In the same fashion, computational grids allow users to plug into a virtual unified resource for their computational needs.

1.3.1 Architecture of a Grid Computing System Grid systems are highly complex and comprise of a host of integrated hardware and software features as illustrated in Figure 1.2. The following sub-sections describe the major components of a grid.

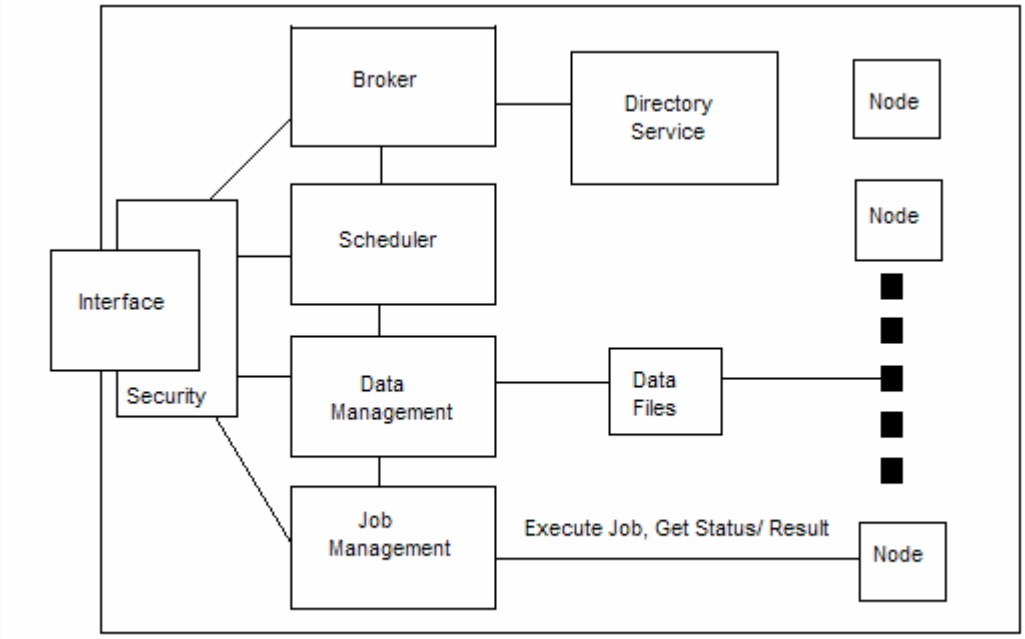

Figure 1.2 Grid Architecture

1.3.1.1 Interface

Grid systems are designed to shield their internal complexities from users. User interfaces can come in many forms and can be application specific. Typically grid interfaces are similar to web portals. A grid portal provides an interface to launch applications which would use its resources and services. Through this interface, users see the grid as a virtual computing resource.

1.3.1.2 Security

Security is a critical issue in grid computing. A grid environment should consist of mechanisms to provide security, which includes authentication, authorization, data encryption etc. Most of the grid implementations include an Open SSL [4] implementation. They also provide a single sign-on mechanism, so that once a user is authenticated, a proxy certificate is created and used while performing actions within the grid.

1.3.1.3 Broker

A grid system typically consists of a diverse range of resources spread across the internet. When a user desires to launch an application through the portal, depending on the application and other parameters provided by the user, the system needs to identify and appropriate the resources to use. This task is accomplished by the grid broker system. The broker makes use of the services provided by the Grid Information Service (GIS) which is also known as the Monitoring and Discovery Service (MDS). It provides information about the available resources within the grid and their status. Upon identifying available resources, the broker needs to choose the most viable resource based on the requirements of the user. Resource brokering is a major research topic in grid computing and forms the focus of what is known as “G-Commerce”.

1.3.1.4 Scheduler

Applications requiring services of a grid could be one large module or could consist of several independent modules with or without data dependencies. Depending on the nature of the application, the scheduler must be able to effectively map the

application or its components onto the best available resource. Most of the grid schedulers use different algorithms to deal with different cases. Grid schedulers have a number of algorithms to choose from depending on scheduling parameters and user requirements. However, the most common criteria for schedulers is to minimize the turnaround time of an application.

1.3.1.5 Data Management

Scheduling high performance applications onto grids constantly requires movement of data files from one node to another. The grid environment should provide a reliable and a secure means for data exchange. The Data Management component of the grid system commonly uses the Grid Access to Secondary Storage (GASS) [4] component to move data files across the grid. The GASS incorporates the GridFTP, which is protocol built over the standard FTP in the TCP/IP protocol suite. The GridFTP protocol adds a layer of encryption and other security features on top of the standard FTP protocol.

1.3.1.6 Job Management

This component includes the core set of services that perform the actual work in a grid environment. It provides service to actually launch a job on a particular resource, check its status, and retrieve results when it is complete. The component is also responsible for ensuring fault tolerance.

1.4 Overview of Task Scheduling in Heterogeneous Computing Environments

There are a number of reasons why scheduling programs or the tasks that comprise the programs is important. For users it is important that the programs they wish to run are executed as quickly as possible (faster turnaround times). On the other hand the owners of computing resources would ideally wish to optimize their machine utilization. These two objectives, faster turnaround times and optimal resource utilization, are not always complementary. Owners are not usually willing to let a single user utilize all their resources (especially in grid systems), and users are not usually willing to wait an arbitrarily long time before they are allowed access to particular resources. Scheduling, from both points of view, is the process by which both the users and the owners achieve a satisfactory quality of service.

1.4.1 Scheduling Strategies

There are different approaches to the selection of processors onto which sub-tasks of a program would be placed for execution. In the static model, each sub-task is assigned to a processor before the execution of a program commences. In the dynamic scheduling model, sub-tasks are assigned to different processors in run-time. In the Hybrid scheduling model, a combination of both static and dynamic scheduling strategies is used.

1.4.1.1 Static Scheduling

In the static model, all sub-tasks of a program are assigned once to a processing element. An estimate of the cost of computation can be made *a priori* . Heuristic models for static task scheduling are discussed in Chapter 2. One of the main benefits of the

static model is that it is easier to implement from a scheduling and mapping point of view. Since the mapping of tasks is fixed a priori, it is easy to monitor the progress of computation. Likewise, estimating the cost of jobs is simplified. Processors can give estimates of the time that will be spent processing the sub-tasks. On completion of the program they can be instructed to supply the precise time that was spent in processing. This facilitates updating of actual running costs and could be used in making performance estimates for new programs. The Static Scheduling model has a few drawbacks. The model is based on an approximate estimation of processor execution times and inter-processor communication times. The actual execution time of a program may often vary from the estimated execution time and sometimes may result in a poorly generate schedule. This model also does not consider node and network failures

1.4.1.2 Dynamic Scheduling

Dynamic scheduling operates on two levels: the local scheduling strategy, and a load distribution strategy. The load distribution strategy determines how tasks would be placed on remote machines. It uses an information policy to determine the kind of information that needs to be collected from each machine, the frequency at which it needs to be collected and also the frequency at which it needs to be exchanged among different machines. In a traditional dynamic scheduling model, the sub-tasks of an application are assigned to processors based on whether they can provide an adequate quality of service. The meaning of quality of service is dependent on the application. Quality of service could mean whether an upper bound could be placed on the time a task needs to wait before it can start its execution; the minimum time under which the task can complete its

execution without interruption and the relative speed of the processor as compared to other processors in the system. If a processor is assigned too many tasks, it may invoke a transfer policy to check to see if it needs to transfer tasks to other nodes and if so, to which ones? The transfer of tasks could be sender initiated or receiver initiated. In the later case, a processor that is lightly loaded will voluntarily advertise to offer its services to heavily loaded nodes.

The main advantage of dynamic scheduling over static scheduling is that the scheduling system need not be aware of the run-time behavior of the application before execution. Dynamic scheduling is particularly useful in systems where the goal is to optimize processor utilization as opposed to minimizing the turnaround times. Dynamic scheduling is also more efficient and fault tolerant when compared to static scheduling.

1.4.1.3 Hybrid Static-Dynamic Scheduling

Static scheduling algorithms are easy to implement and usually have a low schedule generating cost. However, since static scheduling is based on estimated execution costs, it may not always produce the best schedules. On the other hand, dynamic scheduling uses run-time information in the scheduling process and generates better schedules. But dynamic scheduling suffers from very high running costs and may be prohibitively expensive while trying to schedule very large applications with tens and thousands of sub-tasks. Since both the scheduling techniques have their own advantages, researchers have tried to combine them to create a hybrid scheduling technique. Usually in hybrid scheduling, the initial schedule is obtained using static scheduling and the sub-

tasks are mapped onto the respective processors. However, after the execution commences, the processors use run-time information to check and see if the tasks could be mapped to better processors to yield a better a makespan. The running cost of a hybrid scheduling algorithm is greater than the static scheduling algorithms, but is significantly lower than the dynamic only scheduling algorithms.

1.5 NP-complete Problems

Computational problems can be broadly classified into two categories, tractable problems and intractable problems [3]. Tractable problems are the ones whose worst case running time or time complexity is smaller than $O(n^k)$, where n is the input size of the problem and k is a constant. These problems are also known as "Polynomial Time Problems" since they can be executed in polynomial time. The Intractable problems are ones that cannot be executed in polynomial time. They take super-polynomial times to execute.

However, there is a class of problems whose status is unknown to this day. These problems are known as the "NP-complete problems". For these problems, no polynomial time solution has yet been discovered, nor has anyone been able to solve them with a super-polynomial lower bound [3]. Many computer scientists believe that NP-complete problems are intractable. This is mainly because there has been no success in devising a polynomial time solution to any of the existing NP-complete problems so far and if a polynomial time solution is devised for one NP-complete problem, mathematically a polynomial time solution can be devised for all NP-complete problems.

Algorithm designers need to understand the basics and importance of NP-complete problems. If designers can prove that a problem is NP-complete, then there is a good chance that the problem is intractable. If a problem is intractable, it would be better to design an approximation algorithm instead of a perfect algorithm.

The task scheduling problems that form the focus of this dissertation are well known NP-complete problems [3]. We devise approximation algorithms or heuristics to deal with various cases of the task-scheduling problem, which forms the focus of this research.

1.6 Research Objectives

In this dissertation, we intend to propose new algorithms for scheduling tasks in heterogeneous computing systems. In Section 2 we provide a comprehensive literature review on the existing work in the area of task scheduling in heterogeneous computing systems. In Section 3, we propose a new algorithm called the Heterogeneous Critical Node First (HCNF) to schedule a parallel application modeled by a Directed Acyclic Graph (DAG) onto a network of heterogeneous processing elements. In Section 4, we propose a new low-complexity algorithm called the Heterogeneous Largest Task First (HLTF) to schedule independent tasks of a meta-task onto a network of heterogeneous processors. In Section 5, we propose a new algorithm called the Earliest Finish Time with Dispatch Time (EFT-DT) to schedule a set of independent tasks of a meta-task onto a network of heterogeneous processors while also considering the dispatch times. In Section 6, we provide the concluding remarks and also make suggestions for future research in this area.

CHAPTER 2

LIERATURE REVIEW

Among the problems related to task scheduling in heterogeneous computing environments, scheduling a parallel application represented by a directed acyclic graph (DAG) to minimize the overall execution time (makespan) and scheduling a parallel application represented by a meta-task (set of independent tasks) to minimize the makespan are the most important and often researched ones. This section defines the two problems and surveys related research work.

2.1 Scheduling Parallel Applications Represented by Directed Acyclic Graphs onto Heterogeneous Computing Systems to Minimize the Makespan

Many parallel applications consist of sub-tasks with precedence constraints and can be modeled by directed acyclic graphs. This section discusses the problem of scheduling a parallel application represented by a DAG onto a network of heterogeneous processors to minimize its makespan and reviews related research work

2.1.1 Directed Acyclic Graphs

A DAG is represented by $G=\{V,E,W,C\}$. V is the set of n nodes: $\{n_1, n_2, n_3, n_4, ...\}$. E is the set of directed edges of the form (n_i, n_j) which represents an edge directed from

node n_i to n_j. W is the set of node weights of the form w_i, where w_i denotes the weight of node n_i. C is the set of edge weights of the form $c_{i,j}$, where $c_{i,j}$ denotes the weight of the edge (n_i, n_j). A DAG is a graph without a cycle (A directed path from a node onto itself). The set of nodes in a DAG which have an edge directed towards a node n_i are called its *predecessor nodes* and are denoted by $PRED(n_i)$. Likewise, the set of nodes which have a directed edge from a node n_i are called its *successor nodes* and are denoted by $SUCC(n_i)$. Nodes in a DAG that do not have a predecessor are called *start nodes* and nodes that do not have a successor are called *exit nodes*. $blevel(n_i)$ is the bottom level of n_i and is length of the longest path from n_i to any exit node including the weight of n_i. The length of a path in a DAG is the sum of its node and edge weights. $tlevel(n_i)$ is the is the top level of n_i and is the length of the longest path from a start node to node n_i excluding the weight of n_i. The longest path in a DAG is called the *critical path*. A DAG may have multiple critical paths. A sample DAG is illustrated in Figure 2.1. The node weights are to the right of each node and the edge weights are to the left of each edge. Table 2.1 provides the table of values for the sample DAG.

2.1.2 Problem Statement

The objective is to schedule a parallel application represented by a DAG onto a network of heterogeneous processors to minimize its overall execution time. Node-weights in a DAG represent average execution times of nodes over all the processors in the target execution system. Edges represent precedence constraints between nodes. An edge (n_i,n_j) indicates that node n_j cannot start execution until n_i completes execution and

receives all the required data from it. Edge-weights represent the time required to transfer the required data.

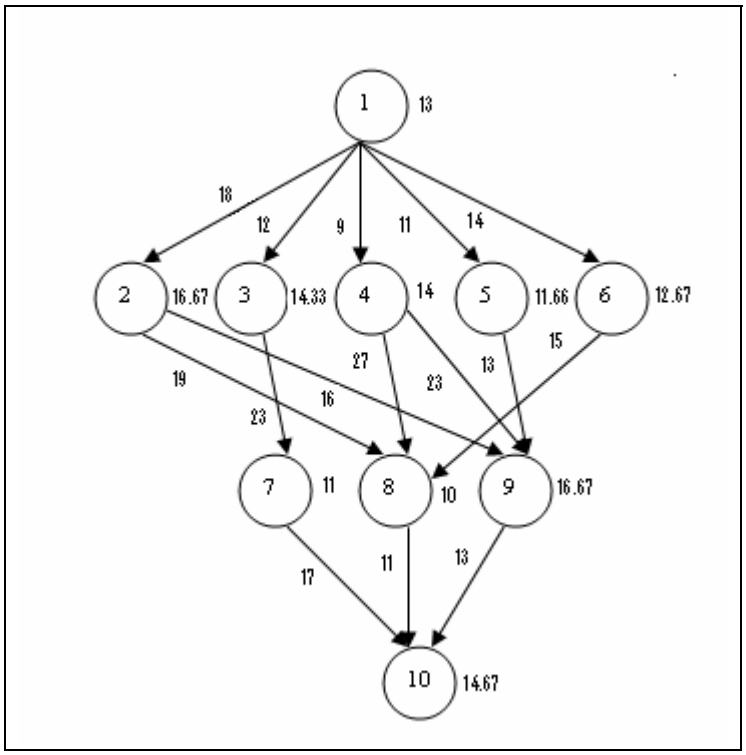

Figure 2.1 A sample DAG, G_1

Table 2.1 Table of values for G_1

n_i	$PRED(n_i)$	$SUCC(n_i)$	$tlevel(n_i)$	$blevel(n_i)$
1	{null}	{2,3,4,5,6}	0	108.01
2	{1}	{8,9}	31	77.01
3	{1}	{7}	25	80
4	{1}	{8,9}	22	81.34
5	{1}	{9}	24	69
6	{1}	{8}	27	63.34
7	{3}	{10}	62.33	42.67
8	{2,4,6}	{10}	66.67	35.67
9	{2,4,5}	{10}	67.67	44.34
10	{7,8,9}	{null}	97.34	14.67

The target execution system consists of a finite number of heterogeneous processors connected with a high speed network. Communication among processors is assumed to be contention-less. Computation and communication is assumed to take place simultaneously. Node-execution is assumed to be non-preemptive; meaning nodes once scheduled on a processor cannot be removed (or preempted) and scheduled on other processors. If a DAG has multiple start nodes, a dummy start node with a zero node weight is added. Zero weight communication edges are then added from the dummy start node to the multiple start nodes. Likewise, if a DAG has multiple exit nodes, a dummy exit node is added. The make-span of a DAG is the time difference between the commencement of execution of the start node and the completion of execution of the exit node. The heterogeneous DAG scheduling problem is NP-complete [28] and can be formally defined as: *To schedule the nodes of a DAG representing a parallel application onto a network of heterogeneous processors such that all the data precedence constraints are satisfied and the overall execution time of the DAG is minimized.* The following sections survey existing research related to this problem.

2.1.3 The Best Imaginary Level Algorithm

The Best Imaginary Level (BIL) algorithm [22] assigns node-priorities based on the *best imaginary level* of each node. At each scheduling step, a free node with the highest priority is selected and mapped onto a processor based on a criterion. Table 2.2 defines the terms used in BIL and Figure 2.2 lists the algorithm.

BIL(n_i, p_j) is the best imaginary level of node n_i on processor p_j. It is the length of the longest path in the DAG beginning with n_i assuming it is mapped onto p_j, and is recursively defined as:

$$BIL(n_i, p_j) = w_{i,j} + \max_{n_k \in Succ(n_i)}[\min(BIL(n_i, p_j), \min_{p \neq j}(BIL(n_i, p_p) + c_{i,p}))].$$

BIL of a node is adjusted to its *basic imaginary make-span* (*BIM*) as follows:

$$BIM(n_i, p_j) = BIL(n_i, p_j) + T\_Available[j].$$

Table 2.2 Definition of terms used in BIL

Term	Definition
N	= $\{n_1, n_2, n_3, n_4, n_5, n_6....\}$//Set of nodes in the DAG, $\|N\|=n$
P	= $\{p_1,p_2, p_3, p_4, p_5, p_6....\}$//Set of processors, $\|P\|=m$
$w_{i,j}$	Time required to execute n_i on p_j
$c_{i,j}$	Time required to transfer all the requisite data from n_i to n_j when they are scheduled on different processors
$BIL(n_i, p_j)$	$= w_{i,j} + \max_{n_k \in Succ(n_i)}[\min(BIL(n_k, p_j), \min_{j \neq l}(BIL(n_i, p_l) + c_{i,k}))]$
$T\_Available[p_j]$	Time at which processor p_j completes the execution of all the nodes previously assigned to it
$BIM(n_i, p_j)$	$= BIL(n_i, p_j) + T\_Avaialble[p_j]$
k	Number of free nodes at a scheduling step
$BIM^*(n_i, p_j)$	$= BIM(n_i, p_j) + w_{i,j} \times \max(k/m - 1, 0)$

If k is the number of free nodes (those nodes whose predecessors have completed execution) at a scheduling step, the priority of a free node is the k^{th} smallest *BIM* value. If the k^{th} smallest *BIM* value is undefined, the largest finite *BIM* value becomes its priority. If two or more nodes have the same priority, ties are broken randomly. At each scheduling step, the free node with the highest priority is selected for mapping. If k is greater than the number of processors, node execution times become more important than the communication overhead. On the contrary, if k is less than the number of available processors, node execution times become less important. The *BIM* value for the selected node is revised to incorporate this factor as follows:

$$BIM^*(n_i, p_j) = BIM(n_i, p_j) + w_{i,j} \times \max(k/m - 1, 0).$$

The processor which provides the highest revised *BIM* value for the node is selected. If more than one processor provides the same revised *BIM* value, the processor that maximizes the sum of the revised *BIM* values of all the other nodes is selected. The time complexity of the algorithm is $O(n^2 + m \log m)$.

2.1.4 The Generalized Dynamic Level Algorithm

The Generalized Dynamic Level (GDL) Algorithm [28] assigns node-priorities based on their *generalized dynamic levels*. A number of factors are incorporated in the calculation of the generalized dynamic level and are explained next. The definition of terms used in GDL is listed in Table 2.3 and the algorithm is listed in Figure 2.3.

BIL Algorithm
ReadyTaskList ← Start node
While *ReadyTaskList* **NOT** empty
  $k \leftarrow |$ *ReadyTaskList*$|$// Number of free nodes
  For all n_i in *ReadyTaskList* and p_j in P
    Compute $BIM(n_i , p_j)$
  End For
  Priority of $n_i \leftarrow k^{th}$ smallest BIM value, or the largest finite *BIM* value if the k^{th} smallest value is undefined
  $n_t \leftarrow$ node in *ReadyTaskList* with the highest priority
  For all p_j in P
    Compute $BIM^*(n_t , p_j)$//Revised *BIM*
  End For
  $p_{fav} \leftarrow$ The processor that provides the highest revised *BIM* value for n_t
  Map n_t on p_{fav}
  ReadyTaskList ← *ReadyTaskList* - n_t + Free nodes(if any)
End While
End BIL

Figure 2.2 The BIL algorithm

$SL(n_i)$ is the static level of a node n_i and is the largest sum of the median execution times of all the nodes from node n_i to an exit node along any path in the DAG. $DL(n_i,p_j) = SL(n_i) - EST(n_i , p_j) + \Delta(n_i , p_j)$ is the Dynamic Level (*DL*) of a node n_i on processor p_j . It indicates how well the node and the processor are matched for execution. Even though $DL(n_i,p_j)$ indicates how well n_i and p_j are matched, it does not indicate how well the descendents of n_i are matched with p_j. $D(n_j)$ is the descendent of node n_i to which n_i passes the maximum data. $F(n_i,D(n_i),p_j) = d(n_i ,D(n_i)) + min_{k \neq j} E(D(n_i),p_k)$ is defined to indicate how quickly $D(n_i)$ can be completed on a processor other than p_j, if node n_i is

executed on processor p_j. The Descendent Consideration (DC) term is defined as: $DC(n_i, p_j) = w^*(D(n_i)) - min \{ E(D(n_i),p_j), F(n_i, D(n_i),p_j)\}$

Table 2.3 Definition of terms used in GDL

Term	Definition
N	$= \{n_1, n_2, n_3, n_4, n_5, n_6....\}$//Set of nodes in the DAG, $\|N\|=n$
P	$= \{p_1,p_2, p_3, p_4, p_5, p_6....\}$//Set of processors, $\|P\|=m$
$w_{i,j}$	Execution time of node n_i on p_j
$c_{i,j}$	Data transfer time from node n_i to n_j
$w^*(n_i)$	Median execution time of n_i over all the processors
$SL(n_i)$	largest sum of the median execution times of all the nodes from node n_i to an exit node along any path in the DAG
$\Delta(n_i, p_j)$	$= w^*(n_i) - w_{i,j}$
$EST(n_i, p_j)$	Earliest start time of n_i on p_j
$DL(n_i, p_j)$	$= SL(n_i)- EST(n_i, p_j) + \Delta(n_i, p_j)$
$D(n_j)$	Descendent node of node n_i to which n_i passes the maximum data
$d(n_i, D(n_i))$	Time required to transfer data from n_i to $D(n_i)$
$E(D(n_i),p_k)$	Time required to execute $D(n_i)$ on processor p_k
$F(n_i, D(n_i),p_j)$	$= d(n_i, D(n_i))+ min_{k \neq j} E(D(n_i),p_k)$
$DC(n_i, p_j)$	$= w^*(D(n_i)) - min \{ E(D(n_i),p_j), F(n_i, D(n_i),p_j)\}$
$C(n_i)$	$= DL(n_i, p_{pref}) - max_{k \neq pref} DL(n_i, p_k)$ (*pref* is the processor on which node n_i obtains the maximum DL)
$GDL(n_i, p_j)$	$= DL(n_i, p_j)+ DC(n_i, p_j)+ C(n_i)$

GDL Algorithm
For all n_i in N
Compute $SL(n_i)$
End For
ReadyTaskList ← *Start Node*
While *ReadyTaskList* is *NOT NULL* **do**
For all n_i in *ReadyTaskList and* p_j in P
Compute $DL(n_i, p_j)$
Compute $DC(n_i, p_j)$
Compute $C(n_i)$
$GDL(n_i, p_j) \leftarrow DL(n_i, p_j) + DC(n_i, p_j) + C(n_i)$
End For
Select the node-processor pair with the maximum *GDL*
Update *ReadyTaskList*
End While
End GDL

Figure 2.3 The GDL algorithm

The preferred processor of a node is the processor which maximizes its dynamic level (DL). The cost of not scheduling a node on its preferred processor is defined as follows.

$C(n_j) = DL(n_i, p_{pref}) - max_{k \neq j} DL(n_i, p_k)$ (p_{pref} is the preferred processor)

The combination of *DL*, the Descendent Consideration (*DC*) term and the cost incurred in not scheduling a node on its preferred processor is used to define the Generalized Dynamic Level (*GDL*) of a node as: $GDL(n_i, p_j) = DL(n_i, p_j) + DC(n_i, p_j) + C(n_j)$.

At each scheduling step, the algorithm selects among the free nodes, the node and the processor with the maximum GDL. The time complexity is $O(n^2 + m \log m)$.

2.1.5 The Levelized Min Time Algorithm

In the Levelized Min Time (LMT) algorithm [16], the input DAG is divided into k levels using the following rules. The levels are numbered *0* to *k-1*. All the nodes in a level

Table 2.4 Definition of terms used in LMT

Term	Definition
N	= $\{n_1, n_2, n_3, n_4, n_5, n_6....\}$//Set of nodes in the DAG, $n=\|N\|$
P	= $\{p_1,p_2, p_3, p_4, p_5, p_6....\}$//Set of processors, $m=\|P\|$
k	Number of levels in the DAG
$w_{i,j}$	Time required to execute n_i on p_j
$c_{i,j}$	Time required to transfer all the requisite data from n_i to n_j when they are scheduled on different processors
$T\_Available[p_j]$	Time at which processor p_j completes the execution of all the nodes previously assigned to it
$EST(n_i, p_j)$	$Max(T\_Available[j], \max_{n_m \in pred(n_i)} EFT(n_m, p_k) + c_{m,i}))$
$EFT(n_i, p_j)$	$= w_{i,j} + EST(n_i, p_j)$

are independent of each other. Level 0 contains the start nodes and level *k-1* contains the exit nodes. For any level j, where $0 < j < k-1$, nodes in level j can have incident edges from any of the nodes in levels *0* thru $j+1$. Additionally, there must be at least one node in level j with an edge incident from a node in level $j+1$. LMT maps the nodes one level at a time starting from level 0. If the number of nodes at a given level is more than the number of processors in the target system, the smallest nodes (based on the average computation times) are merged until the number of nodes equals the number of

processors. Nodes are then sorted by the descending order of their average computations times. At each scheduling, the largest node is mapped onto the processor that provides its minimum finish time. Table 2.4 defines the terms used in LMT and Figure 2.4 lists the algorithm.

LMT Algorithm
Divide the input DAG into *k* levels (level *0* to level *k-1*)
For levels *0* thru *k-1* **do**
num← number of nodes in the current level
If *num*>*m*
Merge the smallest nodes in the current level until *num*=*m*
End If
ReadyTaskList ← Nodes in the current level sorted in the descending order of average node weights
While *ReadyTaskList* is NOT NULL **do**
$n_i \leftarrow$ First node in the *ReadyTaskList*
For all p_j in *P*
Compute $EST(n_i, p_j)$
$EFT(n_i, p_j) \leftarrow w_{i,j} + EST(n_i, p_j)$
End For
Map node n_i on processor p_j which provides its least *EFT*
Update $T\_Available[p_j]$
Update *ReadyTaskList*
End While
End For
End LMT

Figure 2.4 The LMT algorithm

2.1.6 The Heterogeneous Earliest Finish Time Algorithm

The Heterogeneous Earliest Finish Time (HEFT) algorithm [30] assigns node - priorities based on the bottom level (*blevel*) of each node. The *blevel* of a node is the

length of the longest path in the DAG from the node to the exit node. The length of a path in a DAG is the sum of the node and edge weights that constitute the path. At each scheduling step, a node with the highest priority is assigned to a processor that minimizes its finish time. The definition of terms used in HEFT in listed in Table 2.5 and the algorithm is listed in Figure 2.5. As a first step, HEFT traverses the DAG in a top down fashion and computes the *blevels* of all the nodes. At each scheduling step, a node with the highest *blevel* is selected for mapping. Ties are broken randomly

EST(n_i , p_j) is the earliest start time of a node n_i on a processor p_j and is defined as: $EST(n_i, p_j) = Max(T\_Available[p_j], \max_{n_m \in pred(n_i)} EFT(n_m, p_k) + c_{m,i}))$. It is the maximum of a) the time at which processor p_j becomes free or b) The time at which node n_i receives all the required data from its predecessor nodes after the completion of their exeuction. *EFT(n_i , p_j)* is the Earliest Finish Time of a node n_i on a processor p_j and is defined as: *EFT(n_i , p_j) = EST(n_i , p_j) + $w_{i,j}$*. HEFT computes the *EFTs* of the selected node on all the processors and selects the processor that provides the minimum *EFT*. The time complexity is *O(n^2 m)*.

2.1.7 The Critical Path on Processor Algorithm

The critical path is the longest path in a DAG. The length of the critical path gives the lower bound on the overall execution time of the DAG [30]. Minimizing the length of the critical path would aid minimizing the overall execution time of a DAG [30]. The Critical Path on Processor (CPOP) algorithm [30] is a variant of the HEFT algorithm and is from the same authors [30]. CPOP adopts a different mapping strategy for the critical path nodes and the non-critical path nodes.

Table 2.5 Definition of terms used in HEFT

Term	Definition
N	$= \{n_1, n_2, n_3, n_4, n_5, n_6 ....\}$//Set of nodes in the DAG, $n=\|N\|$
P	$= \{p_1, p_2, p_3, p_4, p_5, p_6 ....\}$//Set of processors, $m=\|P\|$
$w_{i,j}$	Time required to execute n_i on p_j
$c_{i,j}$	Time required to transfer all the requisite data from n_i to n_j when they are scheduled on different processors
$priority(n_i)$	$= blevel(n_i)$
$T\_Available[p_j]$	Time at which processor p_j completes the execution of all the nodes previously assigned to it
$EST(n_i, p_j)$	$Max(T\_Available[j], \max_{n_m \in pred(n_i)} EFT(n_m, p_k) + c_{m,i}))$
$EFT(n_i, p_j)$	$= w_{i,j} + EST(n_i, p_j)$

HEFT Algorithm
  For all n_i in N
    Compute $blevel(n_i)$
  End For
  $ReadyTaskList \leftarrow Start\ Node$
    While *ReadyTaskList* is NOT NULL **do**
      $n_i \leftarrow$ node in the *ReadyTaskList* with the maximum *blevel*
        For all p_j in P
          Compute $EST(n_i, p_j)$
          $EFT(n_i, p_j) \leftarrow w_{i,j} + EST(n_i, p_j)$
        End For
        Map node n_i on processor p_j which provides its least *EFT*
        Update $T\_Available[p_j]$ and *ReadyTaskList*
    End While
End HEFT

Figure 2.5 The HEFT Algorithm

Table 2.6 Definition of terms used in CPOP

Term	Definition
N	= $\{n_1, n_2, n_3, n_4, n_5, n_6....\}$//Set of nodes in the DAG, $n=\|N\|$
P	= $\{p_1,p_2, p_3, p_4, p_5, p_6....\}$//Set of processors, $m=\|P\|$
$w_{i,j}$	Time required to execute n_i on p_j
$c_{i,j}$	Time required to transfer all the requisite data from n_i to n_j when they are scheduled on different processors
$priority(n_i)$	$= tlevel(n_i) + blevel(n_i)$
CP processor	$p_j \in P$ which minimizes $\sum_{n_i \in CP} w_{i,j}$ //*CP* is the critical path
$T\_Available[p_j]$	Time at which processor p_j completes the execution of all the nodes previously assigned to it
$EST(n_i, p_j)$	$Max(T\_Available[p_j], \max_{n_m \in pred(n_i)} EFT(n_m, p_k) + c_{m,i}))$
$EFT(n_i, p_j)$	$= w_{i,j} + EST(n_i, p_j)$

CPOP traverses the DAG in a top down fashion to compute the *tlevels* and *blevels* of all the nodes. It identifies the critical path/s and marks the critical path nodes. The priority of each node is the sum of its *tlevel* and *blevel* . At each scheduling step, a free task with the highest priority is selected for mapping. Ties (if any) are broken randomly.

A *CP* processor is defined as the processor that minimizes the overall execution time of the critical path assuming all the critical path nodes are mapped onto it. If the selected node is a critical path node, it is mapped onto the *CP* processor. Else, it is

mapped onto a processor that minimizes its *EFT* (like the HEFT algorithm). The time complexity is $O(n^2 m)$.

CPOP Algorithm
  For all n_i in N
    Compute $tlevel(n_i)$ *and* $blevel(n_i)$
    Identify the critical path/s and mark the critical path nodes
    $priority(n_i) \leftarrow tlevel(n_i) + blevel(n_i)$
  End For
  ReadyTaskList ← *Start Node*
    While *ReadyTaskList* is NOT NULL do
        $n_i \leftarrow$ node in the *ReadyTaskList* with the maximum priority
          If $n_i \in$ critical path
              Map n_i on the *CP* processor
          Else
            For all p_j in P
              Compute $EST(n_i, p_j)$
              $EFT(n_i, p_j) \leftarrow w_{i,j} + EST(n_i, p_j)$
            End For
            Map node n_i on processor p_j which provides its least *EFT*
          End If
          Update $T\_Available[p_j]$
          Update *ReadyTaskList*
    End While
End CPOP

Figure 2.6 The CPOP algorithm

2.1.8 The Fast Critical Path Algorithm

There are three steps involved in a typical static DAG scheduling algorithm: computation of node priorities, node selection, and processor selection. These steps contribute to the overall time complexity of the algorithm. The Fast Critical Path (FCP)

algorithm [24] tries to reduce the overall time complexity by reducing the complexity of the node selection and the processor selection steps.

Table 2.7 Definition of terms used in FCP

Term	Definition
N	= $\{n_1, n_2, n_3, n_4, n_5, n_6....\}$//Set of nodes in the DAG, $n=\|N\|$
P	= $\{p_1,p_2, p_3, p_4, p_5, p_6....\}$//Set of processors, $m=\|P\|$
e	Number of edges in the DAG
$w_{i,j}$	Time required to execute n_i on p_j
$priority(n_i)$	$= blevel(n_i)$
$c_{i,j}$	Time required to transfer all the requisite data from n_i to n_j when they are scheduled on different processors
$T\_Available[p_j]$	Time at which processor p_j completes the execution of all the nodes previously assigned to it
$EST(n_i, p_j)$	$Max(T\_Available[j], \max_{n_m \in pred(n_i)} EFT(n_m, p_k) + c_{m,i}))$
$EFT(n_i, p_j)$	$= w_{i,j} + EST(n_i, p_j)$

Node Selection: FCP tries to reduce the complexity of the node selection process by restricting the size of the *ReadyTaskList* to *m* (The number of processors in the target execution system). Additional free nodes (if any) are stored in a *FIFO* queue. Node priorities are based on their *blevels*. At each scheduling step, a node with the highest priority is selected for mapping. By restricting the size of the *ReadyTaskList* to *m,* the time complexity of the node selection process would be *O(n log m)*.

Processor Selection: The complexity of the processor selection step is reduced by restricting the choice to just two processors: the first processor that becomes free and the *enabling processor* (The processor which is the last to send a data item to a node). The authors [24] prove that the *EFT* of a node is always minimized by one of these two processors. The time complexity of the processor selection step would be reduced to *O(nlogm+e)*. Of the two processors, the one which provides the least *EFT* for the selected node is chosen. The overall time complexity of FCP is *O(nlogm+e)*.

FCP Algorithm
  For all n_i in N
    Compute $tlevel(n_i)$
    $priority(n_i) \leftarrow blevel(n_i)$
  End For
    $ReadyTaskList \leftarrow Start\ Node$
    $AdditionalTaskList \leftarrow NULL$ //FIFO Queue
    While *ReadyTaskList* is NOT NULL do
        $n_i \leftarrow$ node in the *ReadyTaskList* with the maximum priority
          $p_1 \leftarrow$ First processor in P to become free
          $P_2 \leftarrow$ Enabling processor of n_i
          Compute $EST(n_i, p_1)$
          $EFT(n_i, p_1) \leftarrow EST(n_i, p_1) + w_{i,1}$
          Compute $EST(n_i, p_2)$
          $EFT(n_i, p_1) \leftarrow EST(n_i, p_2) + w_{i,2}$
          Map node n_i on processor p_j which provides its least *EFT*
          Update $T\_Available[p_j]$
          Update *ReadyTaskList*
          Update *AdditionalTaskList* (If applicable)
      End While
End FCP

Figure 2.7 The FCP algorithm

2.1.9 The Fast Load Balancing Algorithm

The Fast Load Balancing (FLB) algorithm [24] is a variant of the FCP algorithm. The node selection complexity is reduced by limiting the number of nodes in the *ReadyTaskList* to *m* (number of processors). Additional free nodes, if any, are added to a *FIFO* list. As was discussed in previous section, the earliest start time for a node can be obtained on either the first processor to become free or a task's enabling processor. For each node in the *ReadyTaskList*, the earliest start time of the node on the first processor to become free and the node's enabling processor is calculated. Among the free nodes, the node with the minimum earliest start time is selected and mapped onto the corresponding processor. The overall time complexity of FLB is *O(nlogm+e)*.

Table 2.8 Definition of terms used in FLB

Term	Definition
N	= $\{n_1, n_2, n_3, n_4, n_5, n_6....\}$//Set of nodes in the DAG, $n=\|N\|$
P	= $\{p_1,p_2, p_3, p_4, p_5, p_6....\}$//Set of processors, $m=\|P\|$
$w_{i,j}$	Time required to execute n_i on p_j
$priority(n_i)$	$= blevel(n_i)$
$c_{i,j}$	Time required to transfer all the requisite data from n_i to n_j when they are scheduled on different processors
$T\_Available[p_j]$	Time at which processor p_j completes the execution of all the nodes previously assigned to it
$EST(n_i, p_j)$	$Max(T\_Available[j], \max_{n_m \in pred(n_i)} EFT(n_m, p_k) + c_{m,i}))$
$EFT(n_i, p_j)$	$= w_{i,j} + EST(n_i, p_j)$

```
FLB Algorithm
  For all n_i in N
    Compute tlevel(n_i)
    priority(n_i) ← tlevel(n_i)
  End For
   Readytasklist ← Start Node
   AdditionalTaskList ← NULL // FIFO queue
  While ReadyTaskList is NOT NULL
        For all n_i in Readytasklist
           p_1← First processor in P to become free
           p_2← Enabling processor of  n_i
           Compute EST(n_i , p_1)
           Compute EST(n_i , p_2)
         End For
           Select n_i with the least EST and map it onto the
           corresponding processor.
           Update T_Available[p_j]
           Update ReadyTaskList
           Update AdditionalTaskList (If applicable)
    End While
End FLB
```

Figure 2.8 The FLB algorithm

2.1.10 The Hybrid Re-mapper Algorithm

Static scheduling algorithms use estimates of node execution times in the scheduling process. Estimates can be obtained by techniques such as code profiling and analytical benchmarking [21]. However, actual node execution times may sometimes vary largely from the estimated execution times and may result in a bad schedule. To mitigate this problem, the Hybrid Re-mapper [21] algorithm uses a combination of static mapping and the actual run-time values of node execution times. It tries to fine tune the schedule obtained by a static scheduling algorithm by making use of run-time values as

and when they are made available. The inputs to the algorithm are the DAG and the schedule obtained using a list based static scheduling heuristic. The input DAG is divided into *k* levels marked *0* thru *k-1,* such that nodes in a level do not have precedence constraints between one another. The start nodes are in level *0* and the exit nodes in level *k-1*. Node priorities are based on their *blevels*. Nodes in level *0* are mapped according to the static schedule. For levels *1* thru *k-1*, nodes at a level are considered for re-mapping as soon as the first node of the previous level starts execution. The node with the highest priority is re-mapped onto a processor that provides its least partial completion time (*pct*). In the calculation of partial completion times (see Table 2.9), available run time values (if any) are recursively used. If run time values are not available, statically obtained values are used. The algorithm is listed in Figure 2.9. The time complexity is $O(n^2)$.

Table 2.9 Definition of terms used in Hybird Re-mapper

Term	Definition
N	$= \{n_1, n_2, n_3, n_4, n_5, n_6....\}$//Set of nodes in the DAG, $\|N\|=n$
P	$= \{p_1,p_2, p_3, p_4, p_5, p_6....\}$//Set of processors, $\|P\|=m$
$e_{i,j}$	Time required to execute n_i on p_j in real time
$c_{i,j}$	Time required to transfer all the requisite data from n_i to n_j when they are scheduled on different processors in real time
$priority(n_i)$	$= blevel(n_i)$
$ips(n_i)$	Immediate predecessor set of node n_i
$A[p_j]$	Time at which processor p_j completes the execution of all the nodes previously assigned to it in real time
$dr(n_i)$	$\max_{n_j \in ips(n_i)}(c_{i,j} + pct(n_i, p_k))$
$pct(n_i, p_j)$	$= e_{i,j} + \max(A[j], dr(n_i))$

Hybrid Re-Mapper Algorithm

Divide the input DAG into levels such that nodes in a level are independent of each other

$k \leftarrow$ number of levels

Mark the levels starting with *0* and ending with *k-1*

//Start nodes are in level *0* and exit nodes are in level *k-1*

For all n_i in N

$priority(n_i) \leftarrow blevel(n_i)$

End For

For all n_i in level 0

Map n_i using the static schedule

End For

For levels *1* thru *k-1*

For all *nodes* in the current level

$n_i \leftarrow$ node with the highest priority

For all p_j in P

$dr(n_i) = \max_{n_j \in ips(n_i)}(c_{i,j} + pct(n_i, p_k))$

$pct(n_i, p_j) = e_{i,j} + \max(A[j], dr(n_i))$

End for

Map n_i onto p_j that provides its least *pct*

End For

Update *A[j]*

End for

End Hybrid Re-mapper

Figure 2.9 The Hybrid Re-mapper algorithm

2.1.11 Performance Comparison

The performance of DAG scheduling algorithms depends on a number of factors such as the Communication to Computation Ratio (CCR) (the ratio of the sum of the edge-weights to the sum of the node-weights) of the input DAG, number of nodes,

processor speed variance etc. While running times of an algorithm become significant for large DAGs, it is desirable to have an algorithm with a good performance-complexity tradeoff. The most important performance metric used to compare the performance of DAG scheduling algorithms is the Schedule Length Ratio (SLR). SLR is the ratio of the overall execution time of the input DAG to the sum of the weights of the critical path nodes on the fastest processor. Table 2.9 summarizes the relative performance of the algorithms discussed in the previous sections.

Table 2.10 Comparison of complexity and schedule length ratio of different algorithms

Algorithm *A*	Complexity	Schedule Length Ratio, *L(A)*
BIL	$O(n^2+plogp)$	*L(BIL) < L(GDL)* by 20%
STDS	$O(n^2)$	*L(STDS) < L(BIL)* for CCRs within 0.2 and 1
FLB	*O(nlogp+e)*	*L(HEFT) < L(FLB)* by *63%* when processor speed variance is high. Otherwise FLB performs equally well.
FCP	*O(nlogp+e)*	*L(HEFT) < L(FCP)* by 32 % with high processor speed variance. Otherwise identical.
HEFT	$O(n^2m)$	*HEFT* better than *GDL,LMT* by *8, 52%* respectively.

2.2 Scheduling a Set of Independent Tasks onto a Network of Heterogeneous Processors to Minimize the Overall Execution Time

2.2.1 Problem Statement

Independent tasks are tasks without communication or precedence constraints. A meta-task is a finite set of independent tasks. The overall execution time (make-span) of a meta-task is the time required to complete the execution of all the tasks in it. The target execution system consists of a finite number of heterogeneous processors connected with a high speed network. Tasks in a met-task can have different execution times on different processors. Communication among processors is assumed to be contention-less. Computation and communication is assumed to take place simultaneously. Node execution is assumed to be non-preemptive-nodes once scheduled on a processor cannot be removed (preempted) and scheduled on other processors. The objective of the independent task scheduling problem is formally described as follows. *To schedule the independent tasks of a meta-task onto a network of heterogeneous processors such that the overall execution time of the meta-task is minimized.* In the following sections, existing research work in this area is surveyed.

2.2.2 The Min-Min and Max-Min Algorithms

In the Min-Min algorithm [15], the earliest finish time (EFT) of all the nodes over all the processors is calculated. The node with the least EFT is selected and scheduled

onto the processor on which the minimum EFT was obtained. The process is repeated until all the tasks in the meta-task are scheduled. The time complexity is $O(s^2m)$, where s

Table 2.11 Definition of terms used in Min-Min

Term	Definition
T	$= \{t_1, t_2, t_3, t_4, t_5, t_6....\}$//Meta-Task
s	$=\|T\|$
P	$= \{p_1, p_2, p_3, p_4, p_5, p_6....\}$//Set of processors
m	$=\|P\|$
$w_{i,j}$	Time required to execute t_i on p_j
$EST(t_i , p_j)$	Time at which all the tasks previously assigned to p_j complete execution
$EFT(t_i , p_j)$	$= EST(t_i , p_j) + w_{i,j}$

Min-Min Algorithm
While T is NOT NULL **do**
For all t_i in T and p_j in P
Compute $EFT(t_i , p_j)$
End For
$t_{min} \leftarrow$ task with the least EFT
$p_{min} \leftarrow$ processor providing the least EFT
Map t_{min} on p_{min}
$T \leftarrow T - t_{min}$
End **While**
End **Min-Min**

Figure 2.11 Min-Min Algorithm

is the number of tasks in the meta-task and *m* the number of processors in the target system. The Max-Min algorithm is similar to Min-Max, however; instead of selecting the task with the least EFT, the task with the highest EFT is selected. Min-Min is detailed in Figure 2.11 and the definition of terms used in Min-Min is listed in Table 2.11.

2.2.3 The Sufferage Algorithm

Term	Definition
T	$= \{t_1, t_2, t_3, t_4, t_5, t_6 \ldots.\}$//Meta-Task
s	$=\|T\|$
P	$= \{p_1, p_2, p_3, p_4, p_5, p_6 \ldots.\}$//Set of processors
m	$=\|P\|$
$w_{i,j}$	Time required to execute t_i on p_j
$EST(t_i , p_j)$	Time at which all the tasks previously assigned to p_j complete execution
$EFT(t_i , p_j)$	$= EST(t_i , p_j) + w_{i,j}$
FT_1	Earliest finish time of t_i on any processor p_j
FT_2	Second earliest finish time of t_i on any processor p_j
$Sufferage(t_i)$	$FT_2 - FT_1$

Table 2.12 Definition of Terms used in Sufferage

The Sufferage algorithm [15] is based on the idea that a better mapping of tasks can be obtained by assigning a processor to a task that "suffers" the most in case the task

```
Suffereage Algorithm
  T^l ← temporary set of tasks
  T^l ← NULL
  While T is NOT NULL do
  For all t_i in T and p_j in P
    Compute EFT(t_i , p_j)
    p_temp ← processor on which t_i has the least EFT
      If a task is already assigned to p_temp then
        t_prev ← task already assigned to p_temp
        If Sufferage(t_i) > Sufferage(p_temp) then
          Remove t_prev from p_temp
          Tentatively assign t_i to p_temp
          T ← T - t_i
          T^l ← T^l + t_prev
        Else
          T^l ← T^l + t_i
        End If
      Else
        Tentatively assign t_i to p_temp
          T ← T - t_i
      End If
    End For
    T ← T + T^l
    T^l ← NULL
End While
End Sufferage
```

Figure 2.12 The Sufferage Algorithm

is not assigned to the processor. The sufferage of a task t_i is defined as the difference between the earliest finish time of t_i and the second earliest finish time. Tasks are considered for mapping in an arbitrary order. At each scheduling step, the earliest finish times of a task over all the processors is computed. The processor which provides the

minimum earliest finish time is determined. If a task is already scheduled on it, the suffreages of the task under consideration and the previously scheduled task are compared. If the sufferage of the task under consideration is greater, the previously assigned task is removed and the given task is tentatively assigned to the processor. The removed task is is re-inserted into the meta-task. However, if the sufferage of the task already assigned is greater, the given task is reinserted into the meta-task and is considered for mapping in the next iteration. At the end of the iteration, the tasks which are tentatively mapped onto the processors are permanently mapped. The steps are repeated until all the tasks in the meta-task are mapped. The time complexity is $O(s^2 m)$. Table 2.12 provides the definition of terms used in Sufferage and Figure 2.12 lists the algorithm.

CHAPTER 3

THE HETEROGENEOUS CRITICAL NODE FIRST (HCNF) ALGORITHM

This chapter presents a new task-duplication based static scheduling heuristic called the Heterogeneous Critical Node First (HCNF) for the DAG scheduling problem discussed in section 2.2. The chapter is organized as follows. Section 3.1 discusses the key concepts related to the heterogeneous DAG scheduling problem that motivated the development of HCNF. Section 3.2 discusses the algorithm in detail. Section 3.3 provides the running trace of HCNF. Section 3.4 provides the simulation study and Section 3.5 provides concluding remarks.

3.1 Motivation

The length of the critical path in a DAG provides a lower bound on its overall execution time [30]. Thus, minimizing the execution time of the critical path nodes would abet minimizing the overall execution time of a DAG. One way to achieve this would be to assign top priority to critical path nodes at each scheduling step.

A DAG may have one or more free nodes which are ready to be mapped onto the processors at each scheduling step. In heterogeneous computing environments, local optimization can be obtained at each scheduling step by selecting the largest task among the free nodes and mapping it onto the processor that minimizes its finish time.

Nodes have to wait until they receive all the required data from their predecessors before they could start execution. The predecessor node which is the last to send data to a given node is called the *favorite predecessor*. This process could be potentially expedited by duplicating the execution of favorite predecessors in idle processor times. Duplicating favorite predecessors can potentially suppress communication times and could lead to earlier start times for the nodes.

We propose a static scheduling algorithm called the Heterogeneous Critical Node First (HCNF) that incorporates the strategies discussed above in the scheduling process. At each scheduling step, among the free nodes, HCNF assigns top priority to a critical path node and schedules it onto a processor that minimizes its finish time. In the absence of a critical path node, HCNF picks the largest node and assigns it onto a processor that minimizes its finish time. HCNF also explores the possibility of duplicating favorite predecessors in idle processor times to obtain earlier start times. The algorithm is explained next.

3.2 The HCNF Algorithm

HCNF begins by identifying the critical path/s of the input DAG. Nodes belonging to the critical path/s are marked as *CP* nodes. The algorithm starts the mapping process by mapping the start-node onto the processor that provides its fastest execution time. If the fastest execution time is obtained on more than one processor, the processor with the least average execution time over all nodes is selected. (The average execution time over all nodes of a processor is the sum of the execution times of all the nodes in the DAG on the processor divided by the number of nodes) Among the immeddiate

successors of the start-node, the *CP* node is inserted at the beginning of the *ReadyTaskList*. The remaining nodes are added to the *ReadyTaskList* by the decreasing order of their node weights. At each scheduling step, the first node of the *ReadyTaskList* is selected for mapping. Table 3.1 defines the terms used in HCNF and Figure 3.1 lists the algorithm.

Table 3.1 HCNF-definition of terms

Term	Definition
N	$= \{n_1, n_2, n_3, n_4, n_5, n_6....\}$//Set of nodes in the DAG
n	$=\|N\|$
P	$= \{p_1,p_2, p_3, p_4, p_5, p_6....\}$//Set of processors
m	$=\|P\|$
$w_{i,j}$	Time required to execute n_i on p_j
$c_{i,j}$	Time required to transfer all the requisite data from n_i to n_j when they are scheduled on different processors
$T\_Available[p_j]$	Time at which processor p_j completes the execution of all the nodes previously assigned to it
$pred(n_i)$	Set of immediate predecessors of task n_i
n_{en}	Favorite Predecessor (A node which is the last to send data to a given node.)
$EST_1(n_i, p_j)$	$Max(Max(T\_Available[p_j], Max_{n_m \in pred(n_i)} EFT(n_m, p_k) + c_{m,i}))$
$EST_2(n_i, p_j)$	$Max(Max(T\_Available[p_j], EST(n_{en},p_j)) + w_{en,j}$, $Max_{n_m \in pred(n_i)-n_{en}} EFT(n_m, p_k) + c_{m,i}))$
$EST(n_i, p_j)$	$Min(EST_1(n_i,p_j), EST_2(n_i,p_j))$
$EFT(n_i, p_j)$	$= w_{i,j} + EST(n_i, p_j)$

$EST_2(n_i,p_j)$ is the earliest start time of node n_i on processor p_j assuming that n_{en} (the favorite predecessor of n_i) would be duplicated on p_j. $EST_1(n_i,p_j)$ is the earliest start time of node n_i on processor p_j without duplicating the favorite predecessor. $EFT(n_i,p_j)$ is the earliest finish time of n_i on p_j . At each scheduling step, for the selected node n_i, $EFT(n_i,p_j)$ over all the processors is computed. $fproc(n_i)$ is the processor on which the least *EFT* is obtained. If $EST_2(n_i,p_j)$ is used in the computation of the least *EFT,* n_{en} *is* duplicated on $fproc(n_i)$, otherwise; n_{en} is not duplicated. n_i is mapped onto $fproc(n_i)$. n_i is then removed from the *ReadyTaskList* and its successors are added to it. The nodes in the *ReadyTaskList* are realigned as follows. The *CP* node is inserted at the first position. In the presence of multiple *CP* nodes, the *CP* nodes are sorted by the descending order of their node weights and are inserted at the beginning of the *ReadyTaskList*. All the remaining (non-*CP*) nodes are sorted by the decreasing order of their node weights. The first node in the *ReadyTaskList* is selected for mapping and is scheduled onto a processor that provides its least *EFT* (as discussed earlier). The process is repeated until all the nodes in the DAG are scheduled.

HCNF takes $O(n^2)$ to find the critical path, $O(np)$ to calculate the *EFTs* and $O(n*logn)$ to sort the tasks in the descending order using merge-sort. Ignoring the lower-order terms, the overall time complexity would be $O(n^2)$.

3.3 Running trace of HCNF

The working of HCNF is illustrated with a sample DAG G_1 shown in Figure 3.2. The target execution system consists of three processors: p_1, p_2 and p_3. Node execution-

times are listed in Table 3.2. Node weights in Figure 3.2 represent average execution times. Run-time values for each step of HCNF are shown in Table 3.3. The Gantt chart for the final schedule is shown in Fig. 3.4 and the Gantt chart for the individual steps are shown in Figures 3.5 thru 2.17. HCNF begins by calculating the critical path of G_1 **(** 1→2→9→10) and marking the critical path nodes.

Step 1 (Figure 3.5) The start node (node 1) is mapped onto processor 3 which provides its least finish time of 9 seconds.

Step 2 (Figure 3.6) Among the successors of node 1, the *CP* node (node 2) is inserted at the beginning of the *ReadyTaskList* and the remaining nodes are inserted in the descending order of their weights. Node 2 is selected for mapping and its *EFTs* over all the processors is computed (see Figure 3.3). Both p_1 and p_3 provide the least finish time (27 seconds). However, since the finish time on p_1 is obtained by duplicating node 1, p_3 is chosen. Node 2 is removed and the *ReadyTaskList* is updated to *{3,4,6,5}* .

Step 3 (Figure 3.7) Node 3 is selected for mapping. The minimum *EFT* is obtained on p_1 by duplicating node 1 on p_1. The successor of node 3 (node 7) becomes free as a result of this mapping and the *ReadyTaskList* is updated to *{4,6,5,7}*

Step 4 (Figure 3.7) Node 4 is selected for mapping. The minimum *EFT* is obtained on p_2 by duplicating node 1 on p_2. *The ReadyTaskList* is updated to *{6,5,7}*

Step 5 (Figure 3.8) Node 6 is selected for mapping. The minimum *EFT* is obtained on p_1 .Node 6 is scheduled on p_3 and one of the successor nodes (node 8) becomes free as a result of this mapping. The *ReadyTaskList* is updated to *{5,7,8}*.

Algorithm HCNF

//Identify the *CP* nodes of the input *DAG*

//Map the *Start-Node* onto a processor that provides its fastest execution time

//Among the successors of the *Start-Node* , add the *CP* node to the *ReadyTaskList*

//Add the remaining successors of the *Start-Node* in the decreasing order of task sizes to the *ReadyTaskList*

While *ReadyTaskList* is *NOT NULL* **do**

$n_t \leftarrow$First node in the *ReadyTaskList*

For all $p_j \in P$ **do**

$EST_1(n_t , p_j) = Max\{T\_available[p_j], _{k \neq j} EFT(n_{en} , p_k) + c_{k,j}\}$

If$(EST(n_{en} , p_j) \geq T\_available[p_j])$ **then**

$EST_2(n_t , p_j) = EST(n_{en} , p_j) + w_{en,,j}$

Else

$EST_2(n_t , p_j) = T\_available[p_j] + w_{en,,j}$

End if

If $EST_1(n_t , p_j) \leq EST_2(n_t , p_j)$ **then**

$EST(n_t , p_j) = EST_1(n_t , p_j)$

Else

$EST(n_t , p_j) = EST_2(n_t , p_j)$

Tentatively duplicate n_{en} *on Processor* p_j

End if

$EFT(n_t , p_j) = EST_1(n_t , p_j) + w_{t,j}$

End For

$fproc(n_t) \leftarrow$ processor p_j that provides minimum *EFT* for n_t

Map n_t on $fproc(n_t)$ and permanently duplicate any tentatively duplicated n_{en} node

Add the successors of n_t to the *ReadyTaskList*

ReadyTaskList$\leftarrow$ *ReadyTaskList* - n_t

Realign the *ReadyTaskList* such that the CP node is in the first position and the remaining nodes are sorted in the decreasing order of their weights

End While

End HCNF

Figure 3.1 The HCNF algorithm

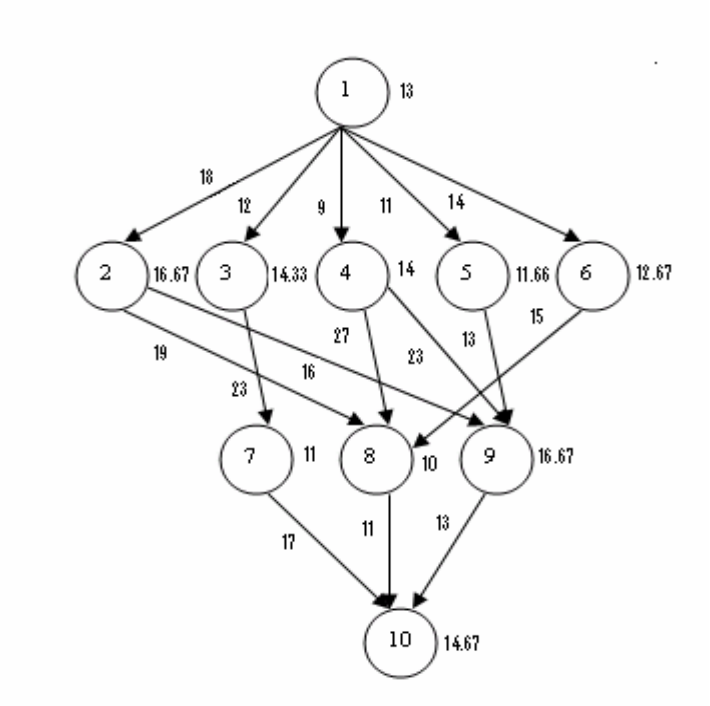

Figure 3.2 Sample DAG *(G_1)*

Table 3.2 Task execution times of G_1 on three different processors

n_i	p_1	p_2	p_3	*Average Execution Time*
1	14	16	9	13
2	13	19	18	16.67
3	11	13	19	14.33
4	13	8	17	14
5	12	13	10	11.66
6	13	16	9	12.67
7	7	15	11	11
8	5	11	14	10
9	18	12	20	16.67
10	21	7	16	14.67

Table 3.3 Run-time values for G_1

Iteration	*ReadyTaskLsit*	n_i	$EST_1(n_i, p_1)$	$EST_1(n_i, p_2)$	$EST_1(n_i, p_3)$	$EFT(n_i)$	$fproc(n_i)$
			$EST_2(n_i, p_1)$ | n_{en}	$EST_2(n_i, p_2)$ | n_{en}	$EST_2(n_i, p_3)$ | n_{en}		
			$EFT(n_i, p_1)$	$EFT(n_i, p_2)$	$EFT(n_i, p_3)$		
1	1	1	0	0	0	9	3
			0 | n/a	0 | n/a	0 | n/a		
			14	16	9		
2	2,3,4,6,5	2	27	27	9	27	3
			14 | 1	16 | 1	n/a | n/a		
			27	35	27		
3	3,4,6,5	3	27	27	27	25	1
			14 | 1	16 | 1	n/a | n/a		
			25	29	46		
4	4,6,5,7	4	25	18	27	24	2
			n/a | n/a	16 | 1	n/a | n/a		
			38	24	44		
5	6,5,7	6	25	24	27	36	3
			n/a | n/a	n/a | n/a	n/a | n/a		
			38	40	36		
6	5,7,8	5	25	24	36	37	1
			n/a | n/a	n/a | n/a	n/a | n/a		
			37	37	46		
7	9,7,8	9	47	50	50	55	2
			50 | 4	43 | 5	47 | 5		
			65	55	67		
8	7,8	7	37	55	48	44	1
			n/a | n/a	68 | 3	55 | 3		
			44	70	49		
9	8	8	51	55	51	56	1
			57 | 6	71 | 6	53 | 4		
			56	66	65		
10	10	10	68	67	68	74	2
			77 | 9	66 | 8	88 | 9		
			89	74	86		

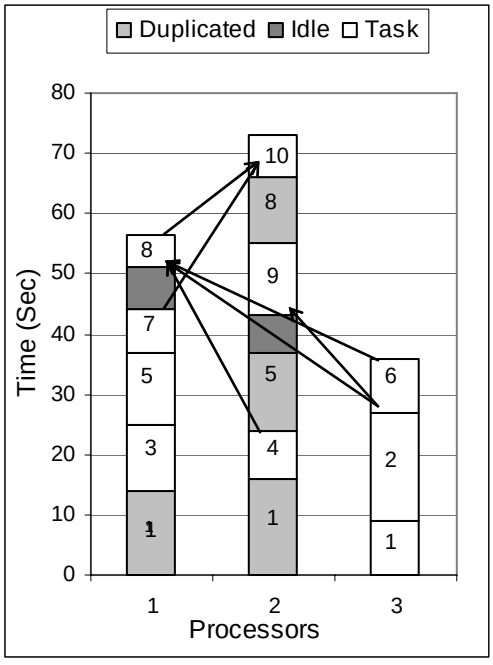

Figure 3.4 Gantt chart for G_1

Step 6 (Figure 3.9) Node 5 is selected for mapping. The minimum *EFT* is obtained on p_1 .Node 5 is scheduled on p_1 and node 9 becomes free as a result of this mapping. The *ReadyTaskList* is updated to *{9,7,8}* (since 9 is a CP node, it is inserted at the beginning of the list)

Step 7 (Figure 3.10) Node 9 is selected for mapping. The minimum *EFT* is obtained on p_2 by duplicating node 5. The *ReadyTaskList* is updated to *{7,8}*.

Step 8 (Figure 3.11) Node 7 is selected for mapping**.** The minimum *EFT* is obtained on p_1. The *ReadyTaskList* is updated to *{8}*

Step 9 (Figure 3.12) Node 8 is selected for mapping. The minimum *EFT* is obtained on p_3. Node 10 becomes free as s result of this mapping and the *ReadyTaskList* is updated to *{10}*.

Step 10 (Figure 3.13) Node 10 is selected for mapping**.** The minimum EFT is obtained on p_2 by duplicating node 8.

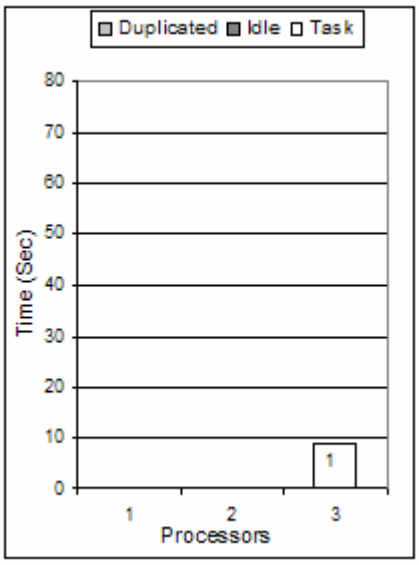

Figure 3.5 HCNF running trace-step 1: Node 1 is scheduled on processor 3

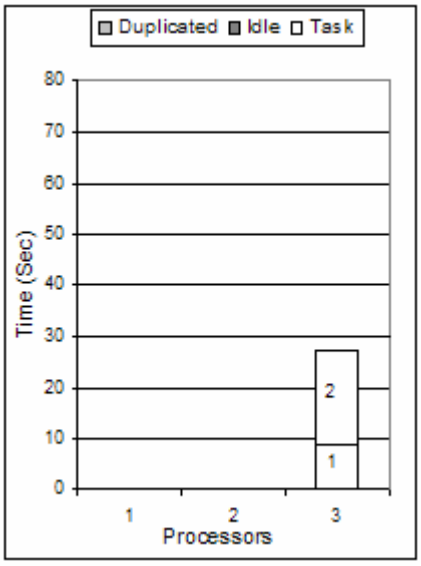

Figure 3.6 HCNF running trace-step 2: Node 2 is scheduled on processor 3

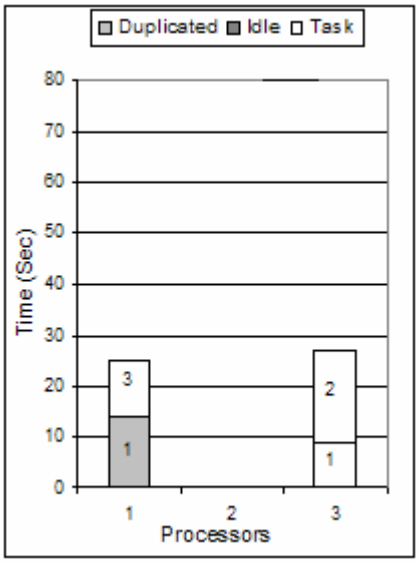

Figure 3.7 HCNF running trace-step 3: Node 3 is scheduled on processor 1 by duplicating node 1

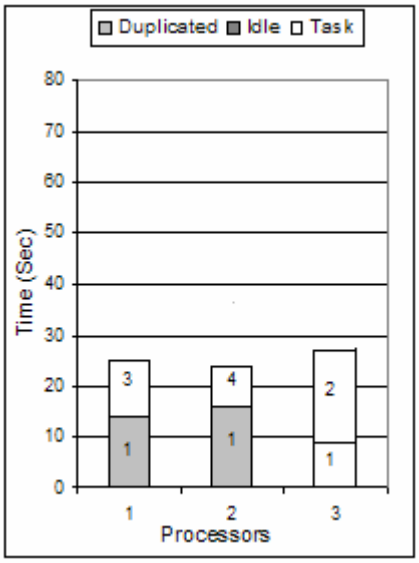

Figure 3.8 HCNF running trace-step 4: Node 4 is scheduled in processor 2 by duplicating node 1

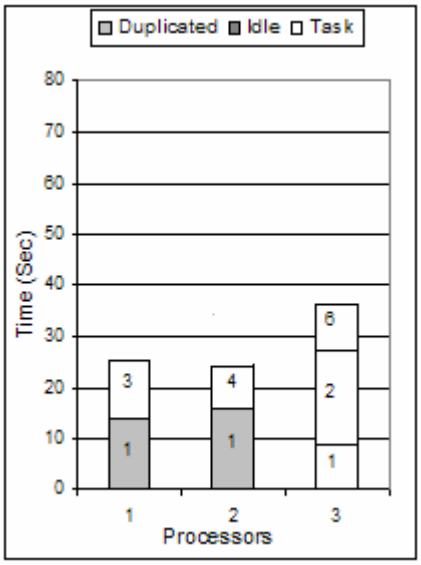

Figure 3.9 HCNF running trace-step 5: Node 6 is scheduled on processor 3

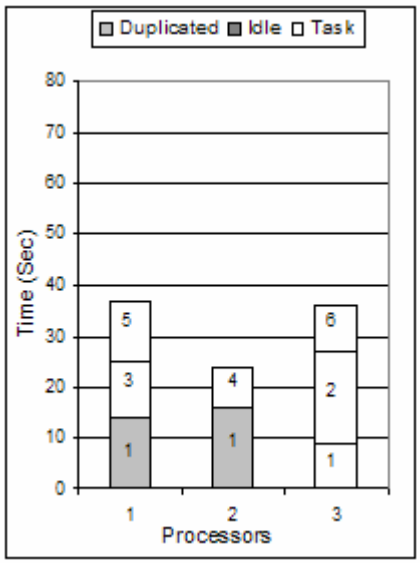

Figure 3.10 HCNF running trace-step 6: Node 5 is scheduled on processor 1

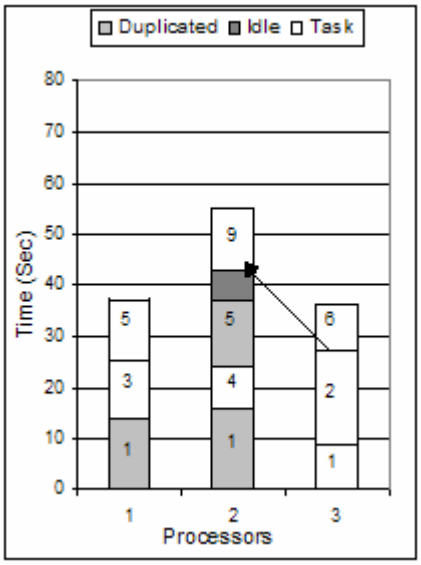

Figure 3.11 HCNF running trace-step 7: Node 9 is scheduled on processor 2 by duplicating node 5

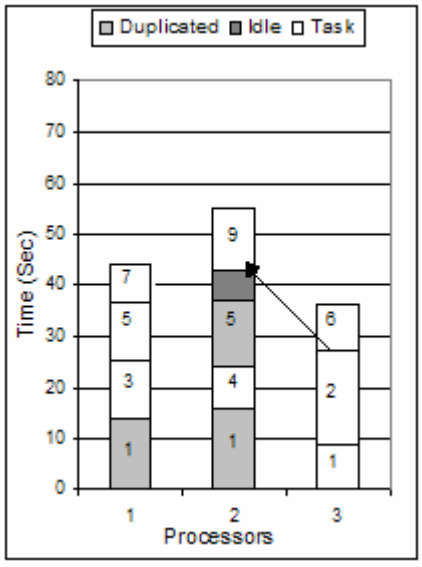

Figure 3.12 HCNF running trace-step 8: Node 7 is scheduled in processor 1

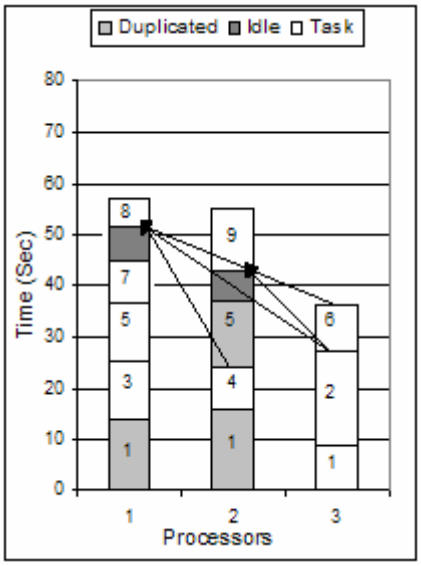

Figure 3.13 HCNF running trace-step 9: Node 8 is scheduled on processor 3

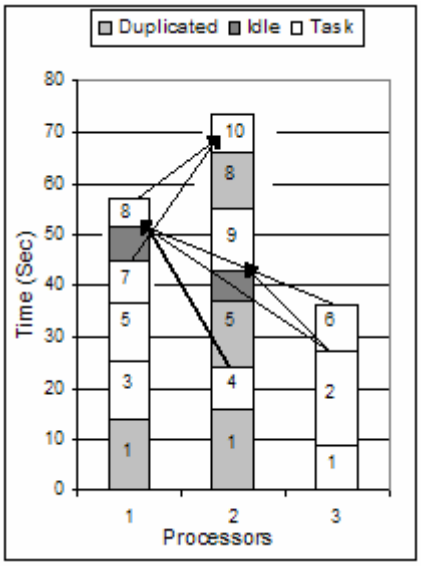

Figure 3.14 HCNF running trace –step10 Node 10 is scheduled on processor 2 by duplicating node 8

3.4 Simulation Study

The simulation study consists of two parts. In the first part, the performance of HCNF is compared against that of the Heterogeneous Earliest Finish Time (HEFT) [30] algorithm. The experimental test suite[18] includes: randomly generated graphs, Gaussian elimination graphs, Trace graphs, Benchmark graphs and Application graphs. In the second part, a parametric random graph generator is developed to generate a diverse range of graphs with specified input parameters. The performance of HCNF is compared against that of the HEFT and the Scalable Task Duplication based scheduling algorithm (STDS) [25].

3.4.1 Performance Parameters

The three commonly used performance parameters to gauge the performance of DAG scheduling algorithms are:

Schedule Length Ratio (SLR): The ratio of the overall execution time of a DAG to the sum of the weights of its critical-path nodes on the fastest processor.

Speedup: The ratio of the sequential execution time of the DAG on the fastest processor to the parallel execution time.

Efficiency: The ratio of the speedup to the number of processors in the system.

3.4.2 Randomly Generated Graphs

The performance of HCNF and HEFT was compared using randomly generated graphs of different sizes and CCRs. Each node in the random graph was allowed to have up to five children. Node and the edge weights were generated randomly and the edge weights were then iteratively adjusted to obtain a given CCR.

The SLR and speedup of HCNF and HEFT was compared using graphs of different sizes. For each graph size shown in Figures 3.15 and 3.16, readings were averaged using 10 random graphs of the same size with CCRs ranging from 0.5 to 1.5. and *out_degree* = {1,2,5,100}. The average SLR of HCNF was better than HEFT by 12.3% and the speedup was better than HEFT by 7.9 %.

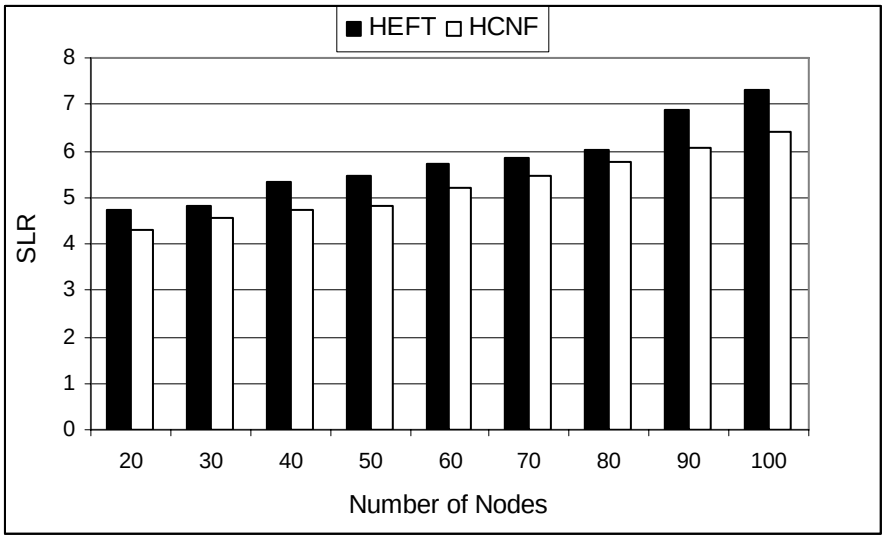

Figure 3.15 Average SLR vs. number of nodes

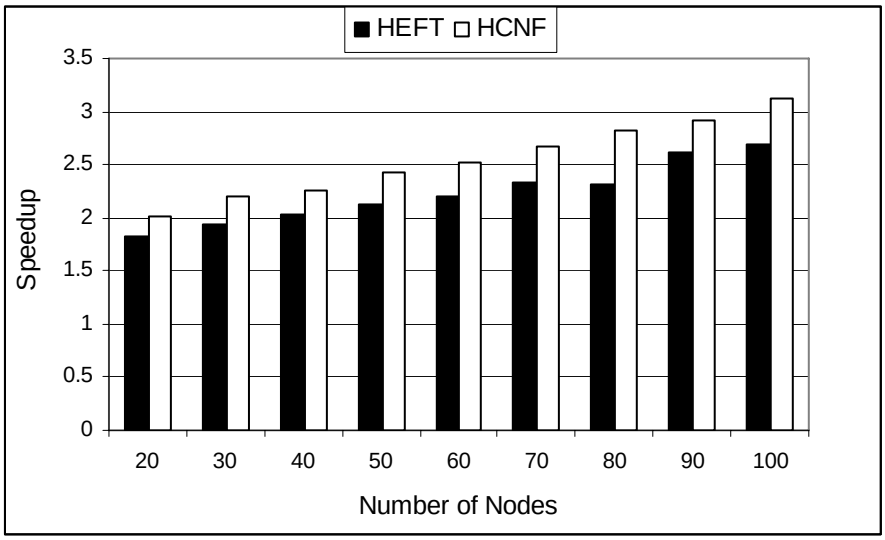

Figure 3.16 Average speedup vs. number of nodes

3.4.3 Gaussian Elimination Graphs

The SLR and Efficiency of HEFT and HCNF were compared using DAGs representing the Gaussian Elimination algorithm. Figure 3.17 gives the SLR for matrix sizes ranging from 5 to 15. HCNF outperformed HEFT by an average of 25.7%. Figure 3.18 gives the efficiency for different number of processors, with the matrix size fixed at 50. HCNF outperformed HEFT by an average of 22.6%. The efficiency of HCNF increased with the number of processors because of increased speedup facilitated by enhanced task duplication (in the presence of a lager number of processors).

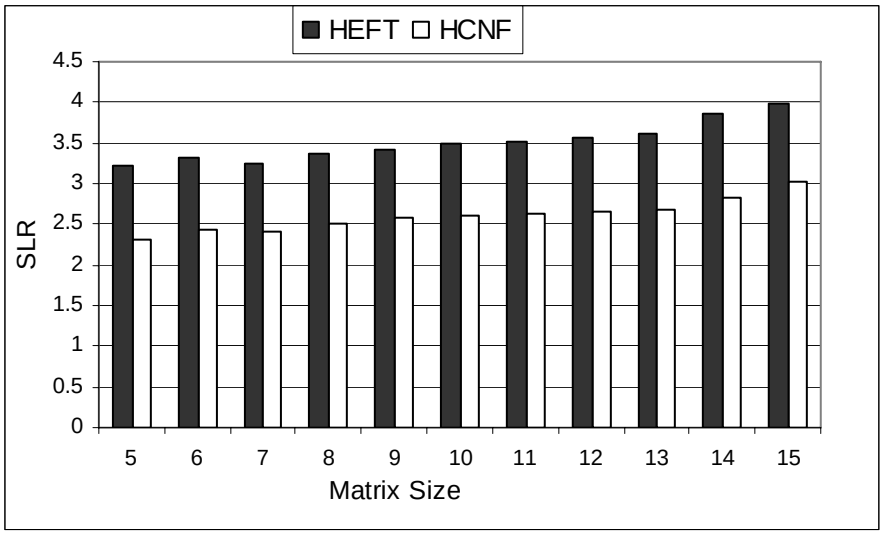

Figure 3.17 Average SLR vs. matrix size

3.4.4 Benchmark Graphs

DAG scheduling algorithms are commonly compared using randomly generated graphs. However, to facilitate a fair and an unbiased comparison of algorithms from

different authors, some researchers [21] have proposed using benchmark graphs. In the following sections we compare the performance of HCNF using the "benchmark graph test suite" [21]. The benchmark test suite consists of: Trace graphs, Graphs with optimal solution generated by the branch and bound technique, Graphs with predetermined optimal solutions and Application graphs

3.4.4.1 Trace Graphs

These graphs are obtained from the referenced articles listed in Table 3.4.The SLR and speedup of HEFT and HCNF was compared using these graphs. Figures 3.19 and 3.20 show the results. HCNF outperformed HEFT in SLR and speedup by an average of 29.5% and 38.4% respectively

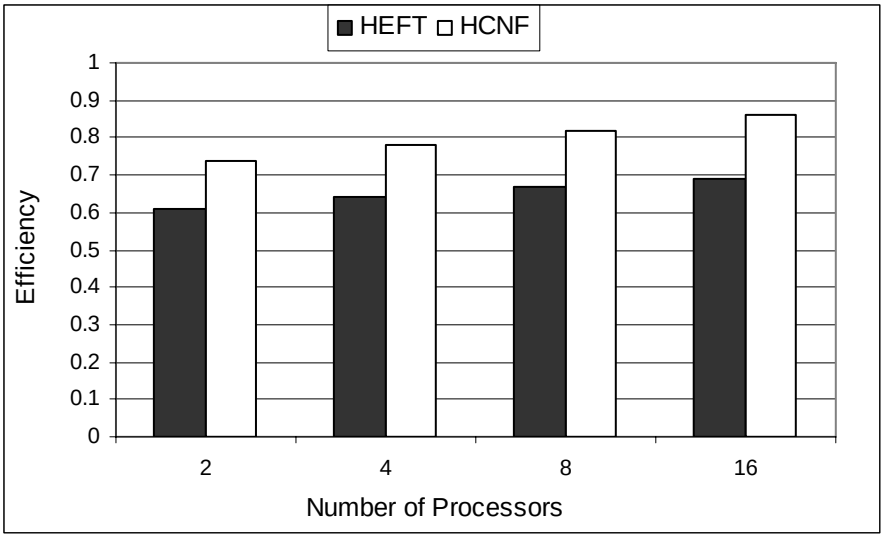

Figure 3.18 Efficiency vs. no. of processors

.3.4.4.2 Random Graphs with Optimal Solutions (RGBOS)

DAGs in this set are small sized, with the maximum node size being 32. Their optimal solution can be obtained using the branch and bound technique. The set consists of three subsets of graphs with different CCRs (0.1, 1.0, 10.0), and number of nodes vary from 10 to 32, in increments of 2. Figures 3.21 thru 3.26 show the results. HNCF outperformed HEFT in SLR and speedup by 32.5% and 24.6% respectively.

3.4.4.3 Random Graphs with Pre-Determined Optimal Schedules (RGPOS)

The graphs in this set are reverse engineered [23]. A schedule for a set of multiprocessors is generated and then the node and the edge weights are generated randomly, but, consistent with the generated schedule. The graphs comprise of three sets with CCR values 0.1,1.0 and 10.0. Within each set, the number of nodes vary from 50 to 500 in increments of 50. Figures 3.27 thru 3.32 show the results. HCNF outperformed HEFT in SLR and speedup by 21.1% and 16.9% respectively.

Table 3.4 Trace graph details

Graph Tag	Trace Graph	# of Nodes	Article Reference
D1	Ahmed-Kwok	13	[22]
D2	Yang-1	7	[13]
D3	Colin-Chretienne	9	[25]
D4	McCreary	9	[12]
D5	Kruatrachue	11	[16]
D6	Yang-2	7	[19]
D7	Ranka	11	[22]
D8	Shirazi	11	[23]
D9	Wu-Gajski	18	[25]
D10	Al-Maasarani	16	[33]
D11	AL-Mouhamed	17	[32]

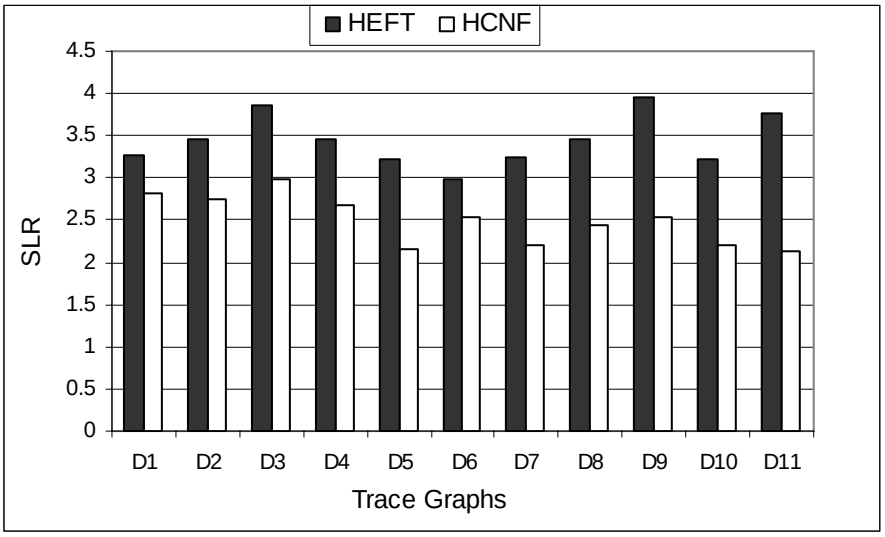

Figure 3.19 Trace Graphs-SLR

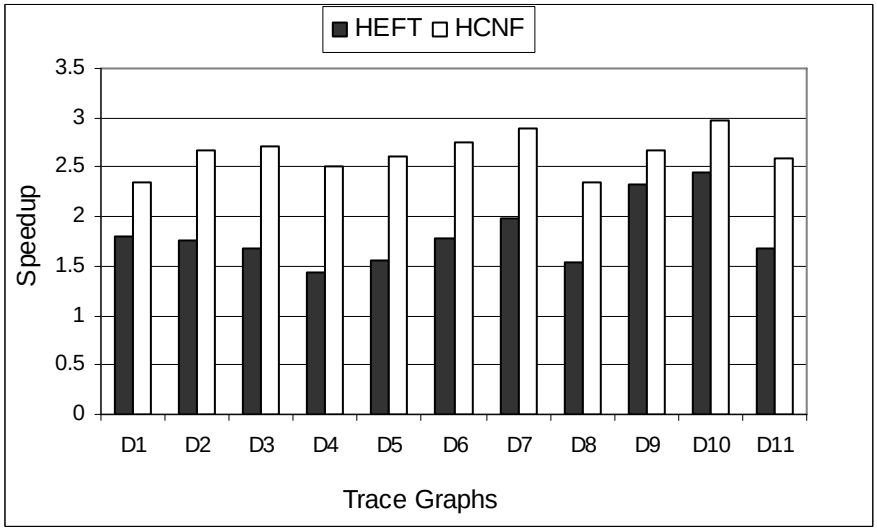

Figure 3.20 Trace Graphs-Speedup

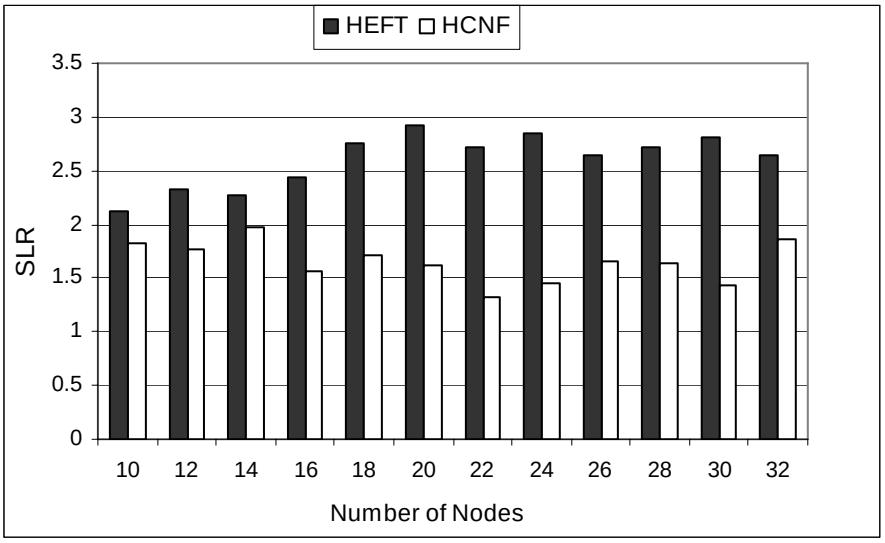

Figure 3.21 RGBOS SLR (CCR = 0.1)

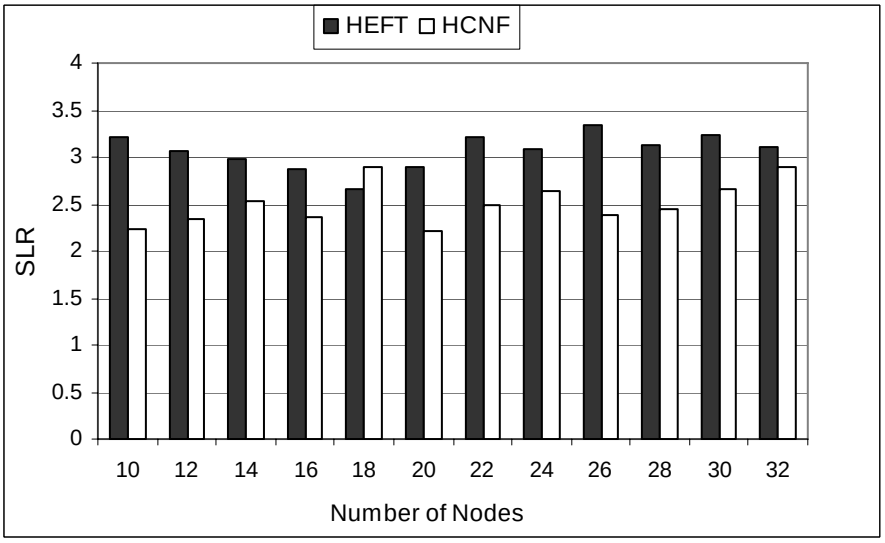

Figure 3.22 RGBOS SLR (CCR = 1.0)

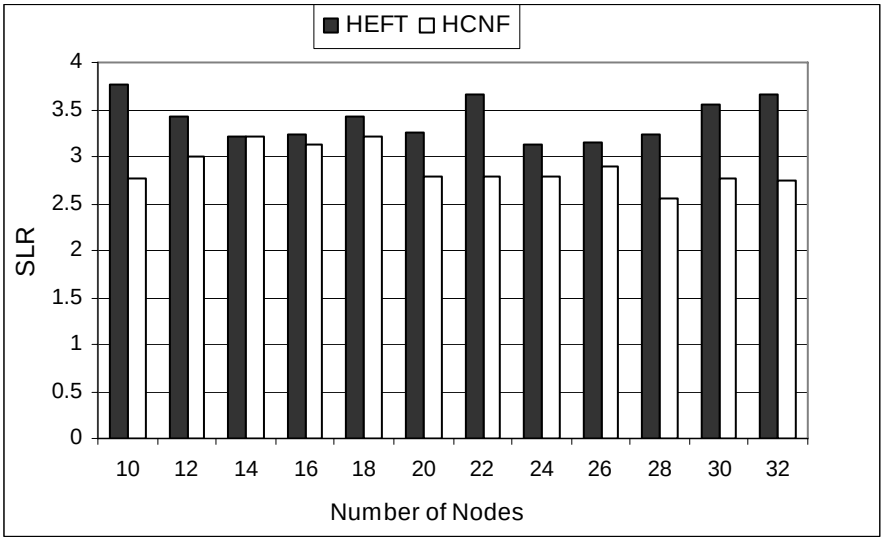

Figure 3.23 RGBOS SLR (CCR = 10.0)

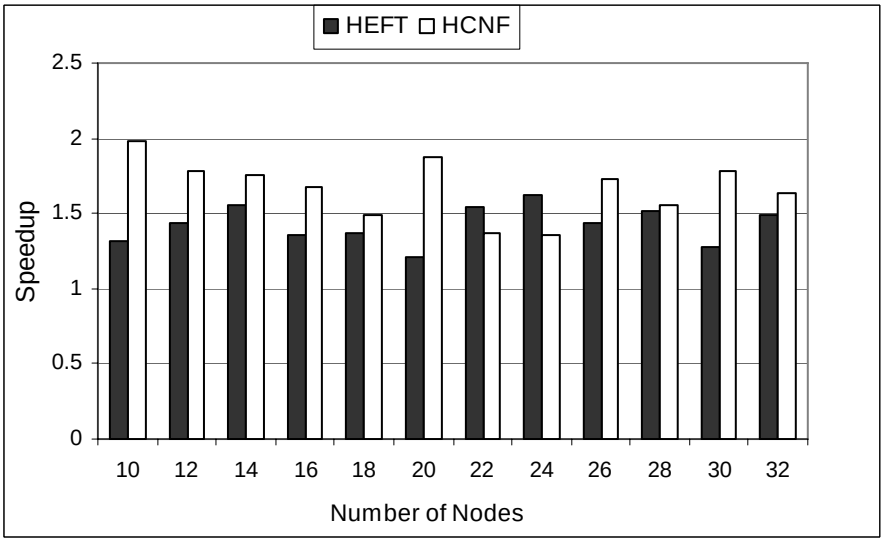

Figure 3.24 RGBOS Speedup (CCR = 0.1)

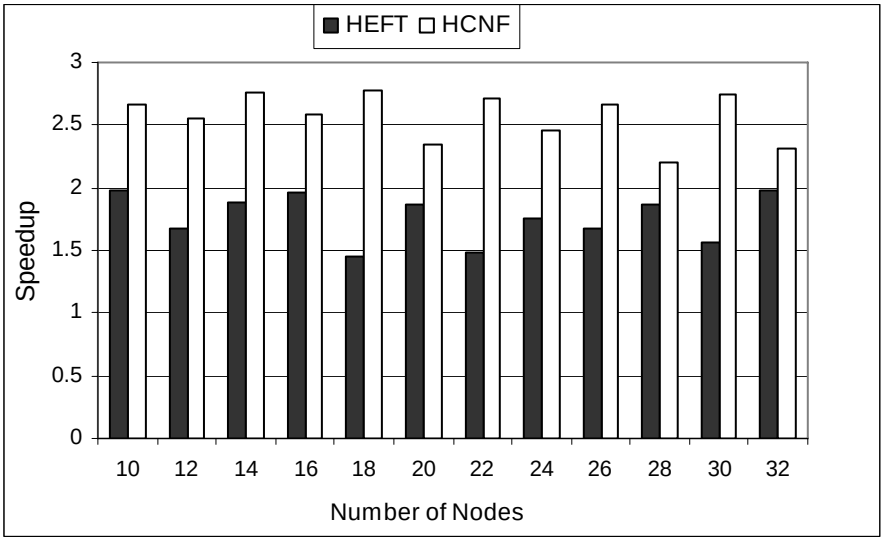

Figure 3.25 RGBOS Speedup (CCR = 1.0)

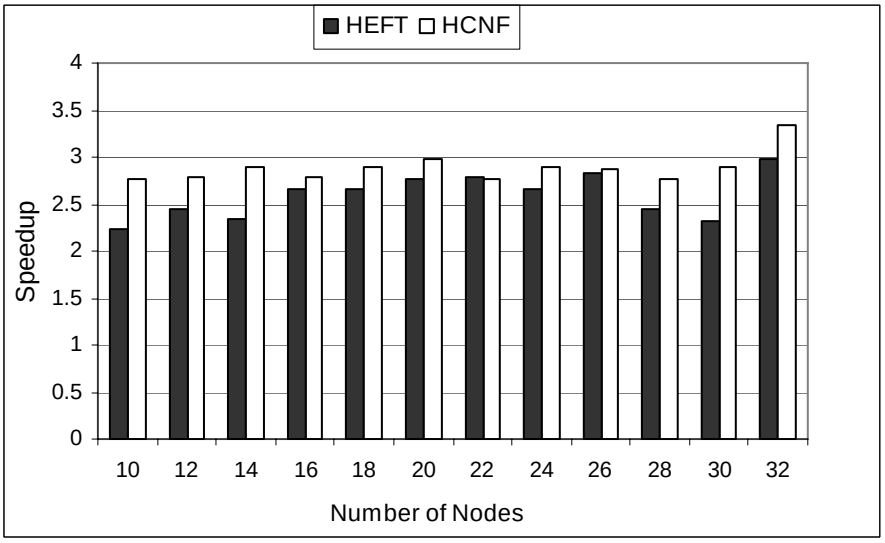

Figure 3.26 RGBOS Speedup (CCR = 10.0)

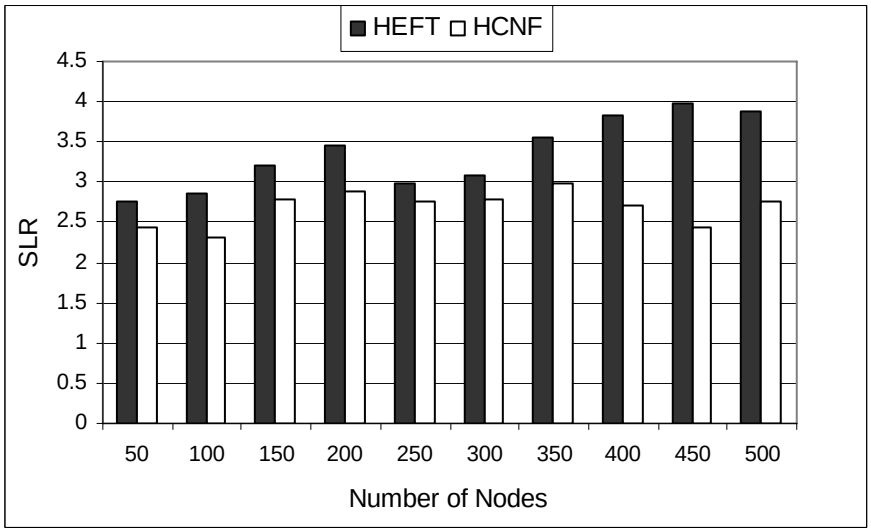

Figure 3.27 RGPOS SLR (CCR = 0.1)

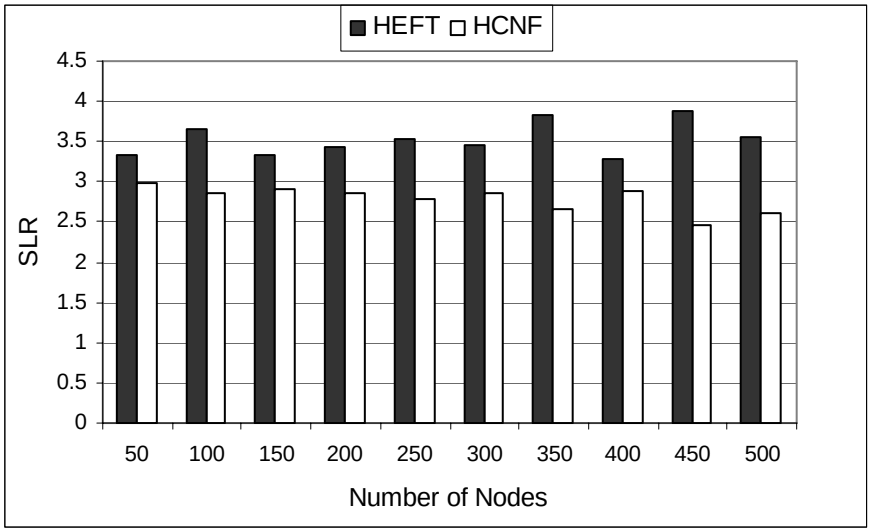

Figure 3.28 RGPOS SLR (CCR = 1.0)

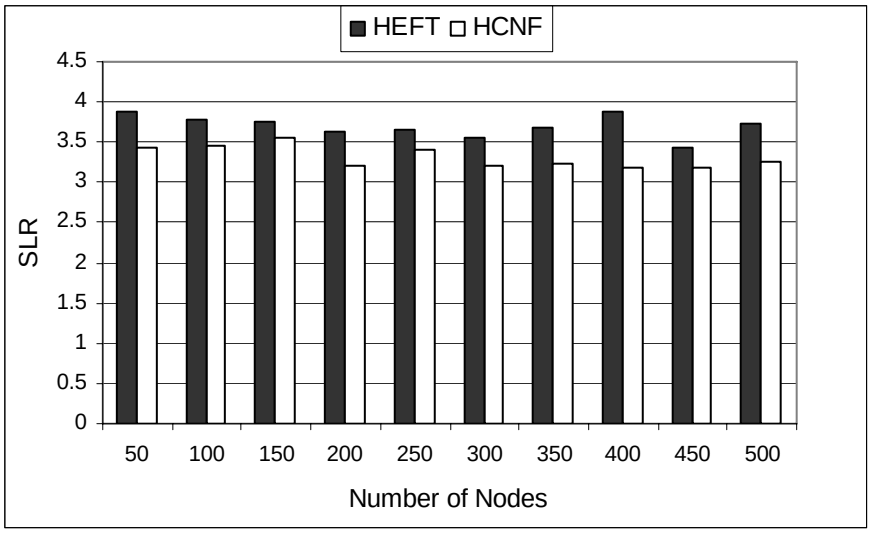

Figure 3.29 RGPOS SLR (CCR = 10.0)

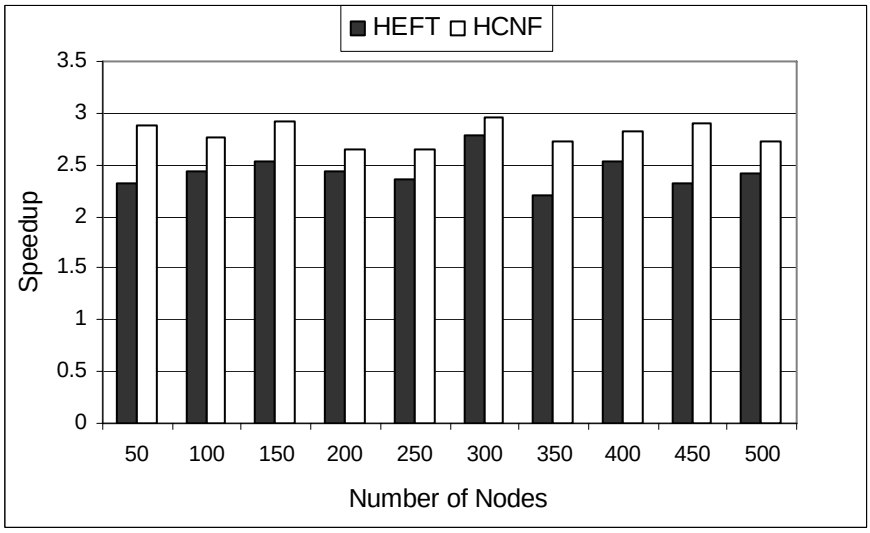

Figure 3.30 RGPOS Speedup (CCR = 0.1)

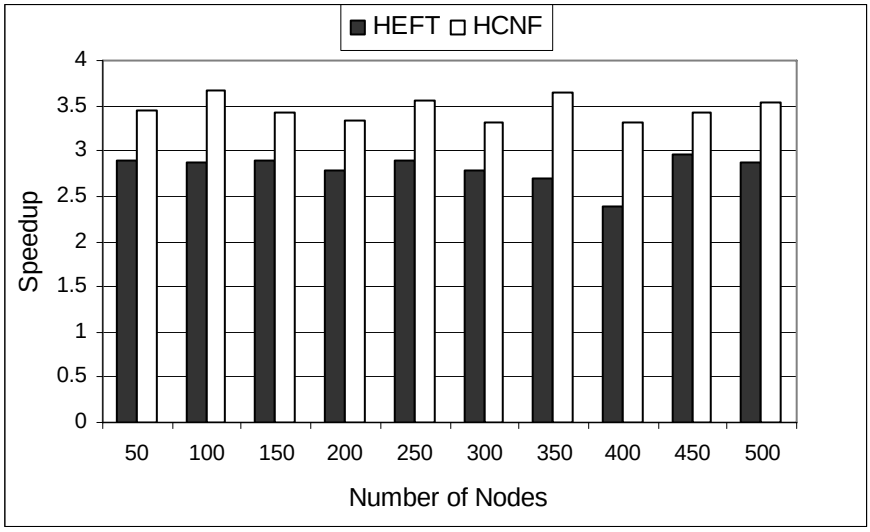

Figure 3.31 RGPOS Speedup (CCR = 1.0)

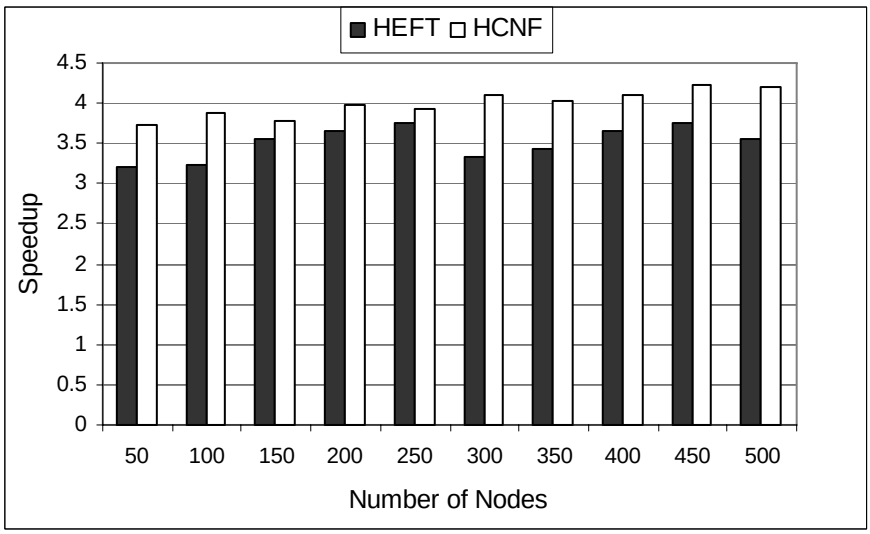

Figure 3.32 RGPOS Speedup (CCR = 10.0)

3.4.4.5 Application Graphs

These graphs represent a few numerical parallel application programs. This set contains of over 320 graphs in six categories: Cholesky factorization, LU decomposition, Gaussian elimination, FFT, Laplace transforms and Mean Value Analysis (MVA). The number of nodes ranges from 100 to 300. Figures 3.33 to 3.44 show the results of the simulation. On an average, HCNF outperformed HEFT in SLR and Speedup by 27.5% and 22.7% respectively

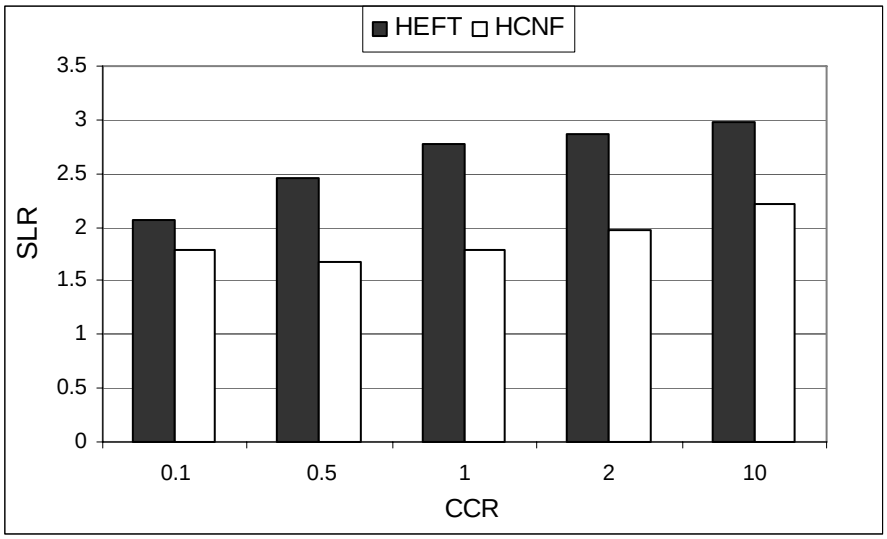

Figure 3.33 Fast Fourier Transform SLR vs. CCR

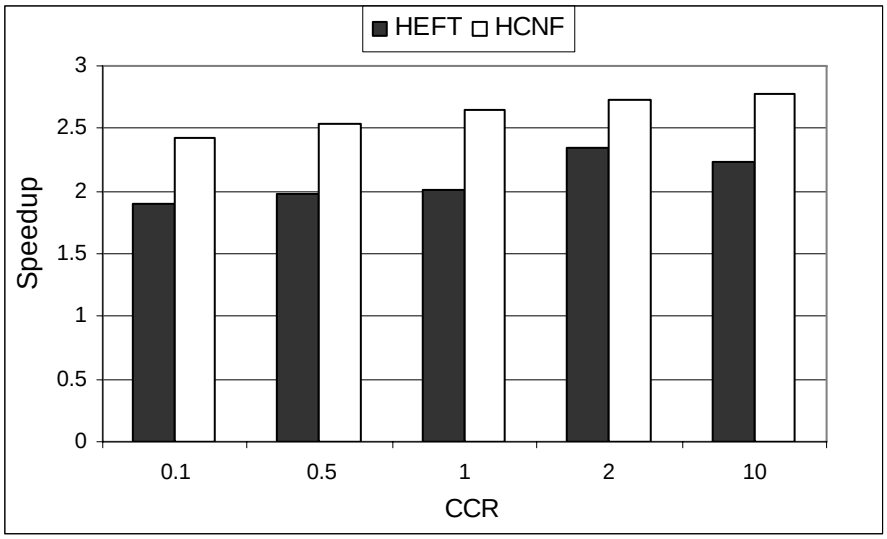

Figure 3.34 Fast Fourier Transform Speedup vs. CCR

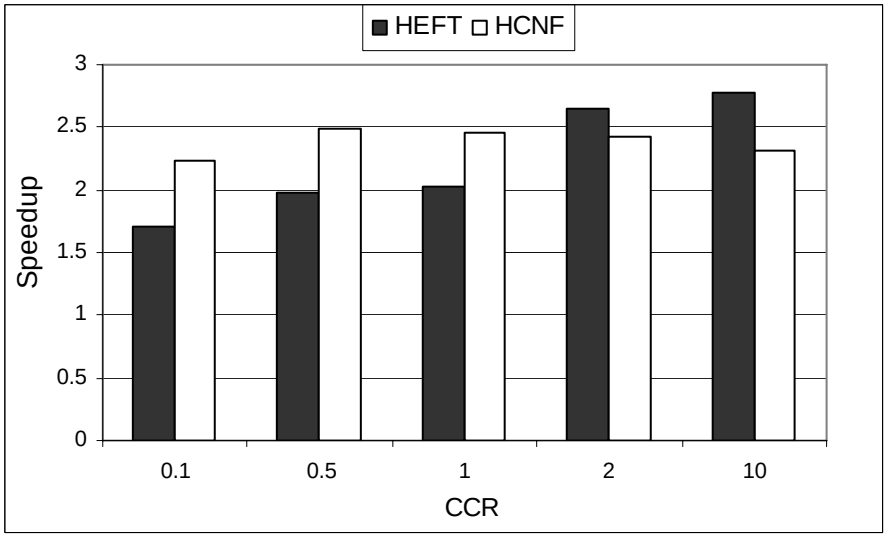

Figure 3.35 Cholesky Factorization Speedup vs. CCR

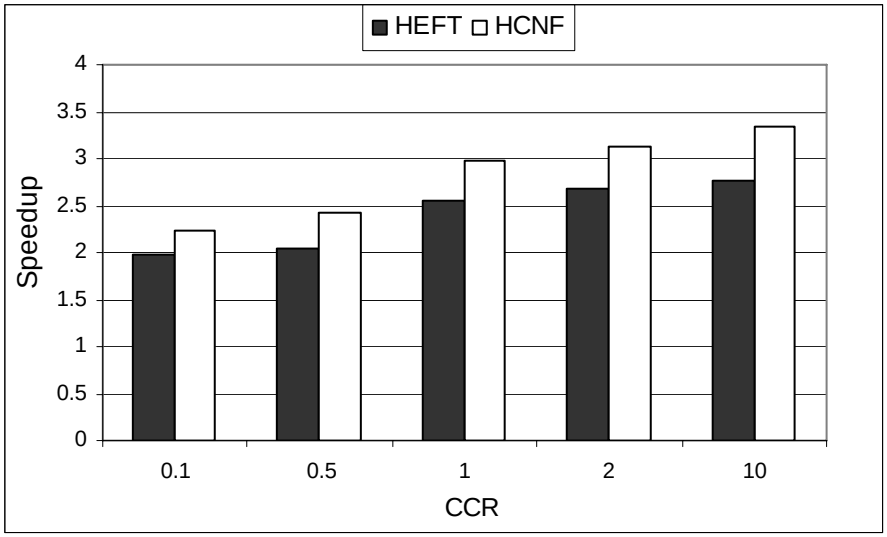

Figure 3.36 Gaussian Elimination Speedup vs. CCR

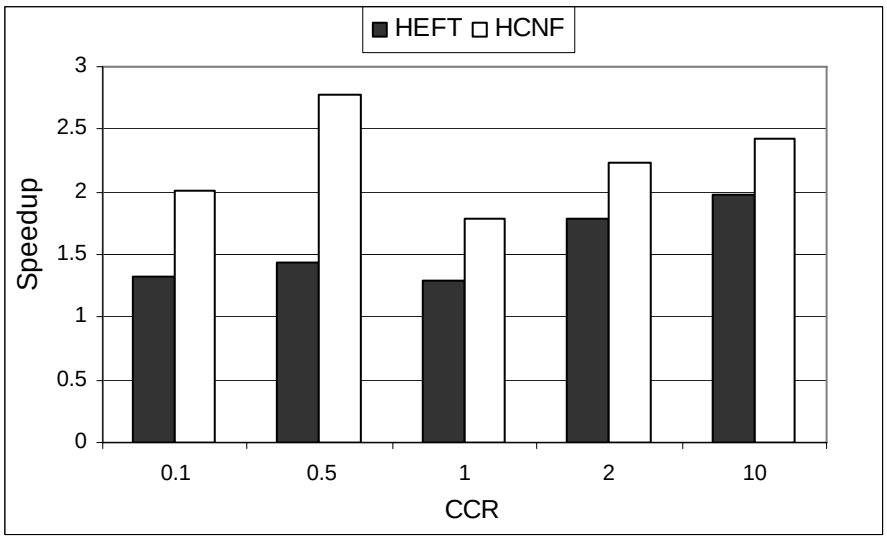

Figure 3.37 Laplace Transform Speedup vs. CCR

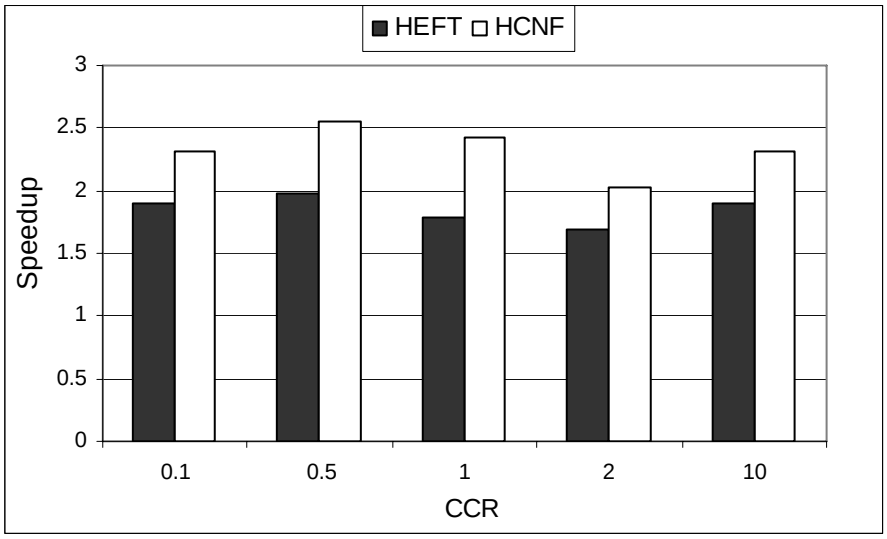

Figure 3.38 LU Decomposition Speedup vs. CCR

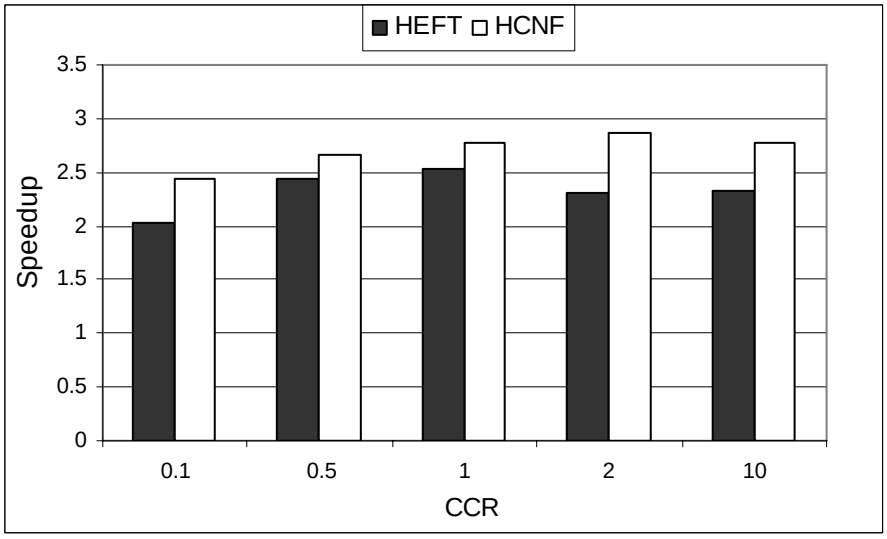

Figure 3.39 MVA Speedup vs. CCR

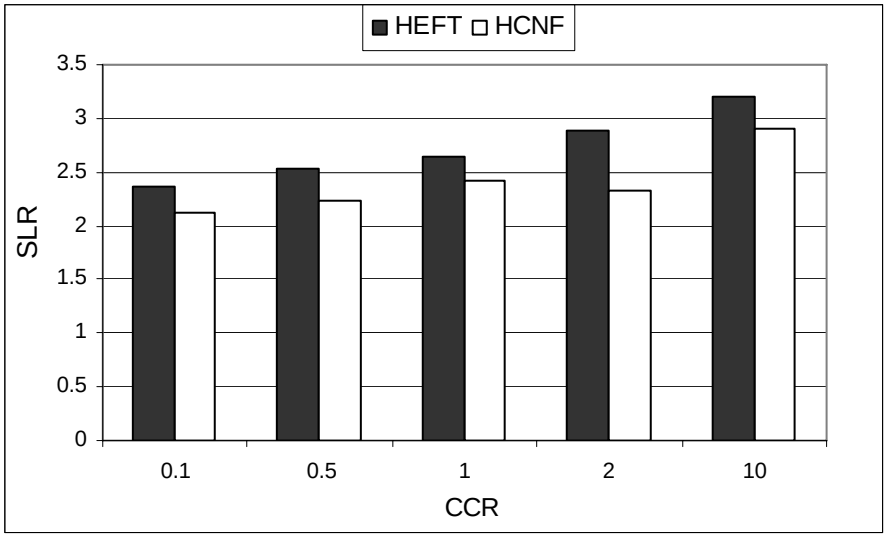

Figure 3.40 Cholesky SLR vs CCR

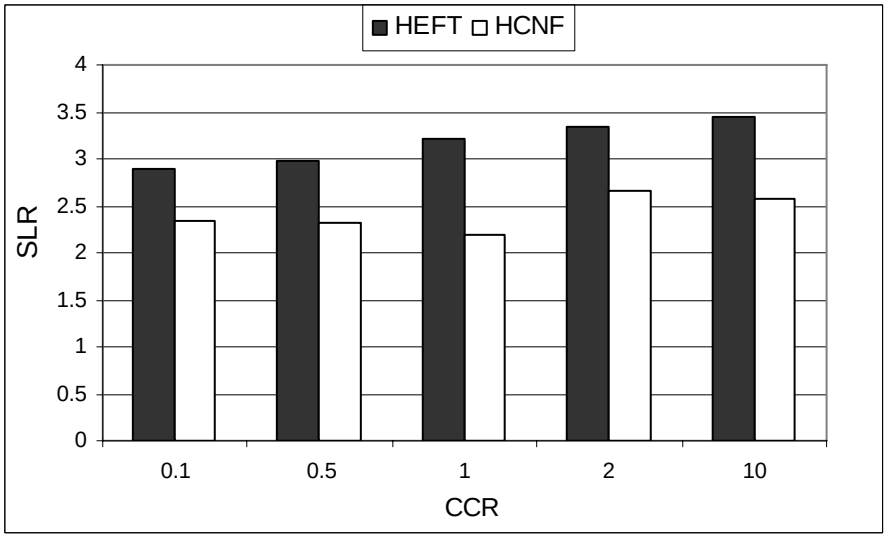

Figure 3.41 Gaussian Elimination SLR vs.CCR

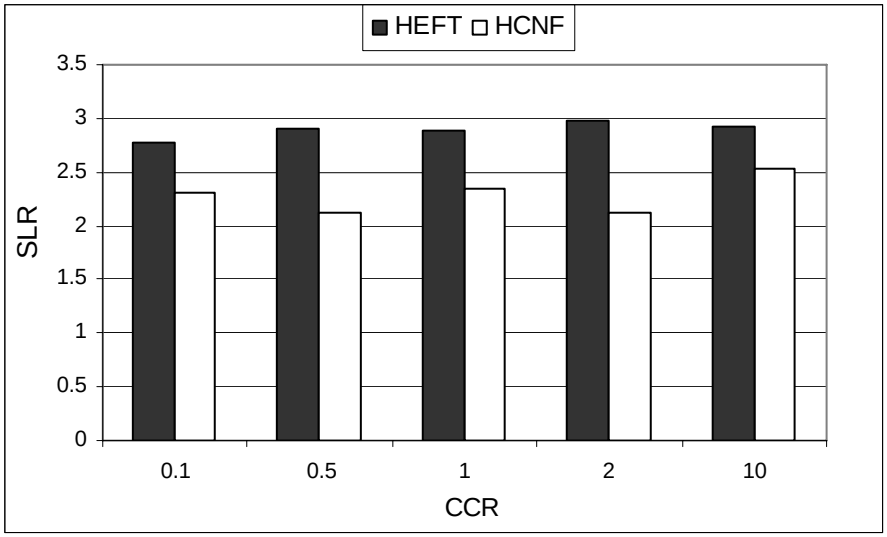

Figure 3.42 Laplace Transform SLR vs.CCR

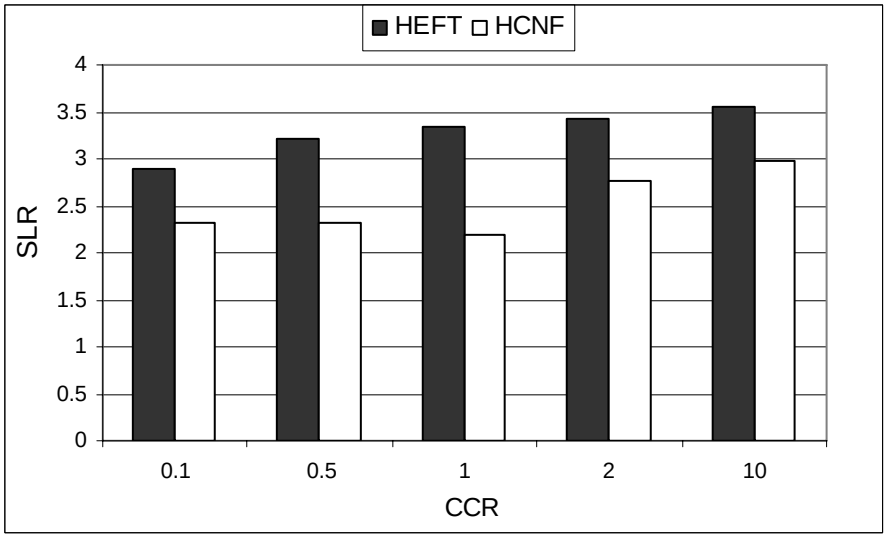

Figure 3.43 LU Decomposition SLR vs. CCR

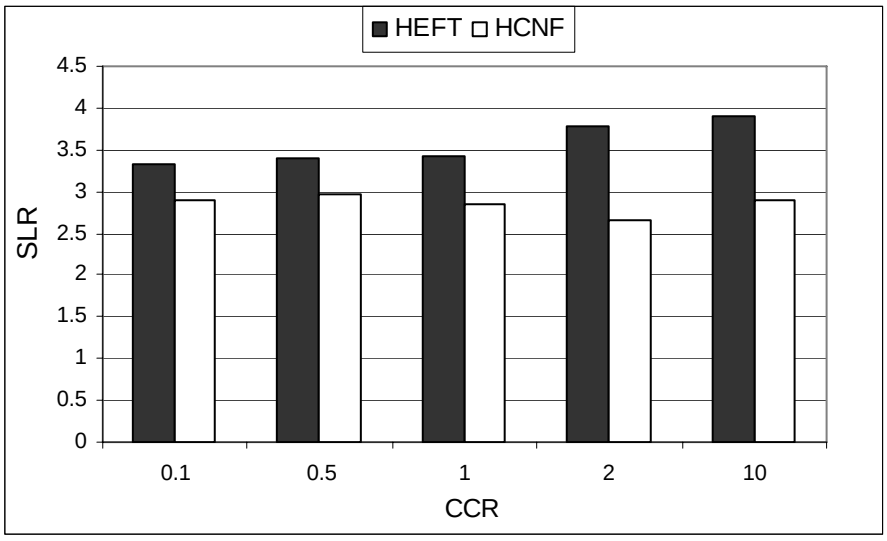

Figure 3.44 MVA SLR vs. CCR

3.4.5 Performance Comparison using a Parametric Random Graph Generator

The performance of DAG scheduling algorithms varies largely with the type of the input DAG. In order to conduct a fair comparison, we need to evaluate the performance using a comprehensive set of randomly generated DAGs, exhibiting a wide range of parameters. In our simulation study, a parametric random graph generator was developed to generate diverse DAG types based on the following input parameters.

- n : Number of nodes in the DAG

- *CCR* (Communication to Computation Ratio): Ratio of the sum of the edge weights to the sum of the node weights in a DAG.
- *out_degree*: Maximum number of children a node in the DAG can have.
- α (The shape parameter of a DAG) : The height of a DAG is randomly generated from a uniform distribution with mean equal to $\alpha \times \sqrt{n}$. The width of a DAG is randomly generated from a uniform distribution with mean equal to $\sqrt{n} \div \alpha$
- β (Range percentage of computation costs on processors): If the average computation cost of a node over all the processors is *avg_comp* , the computation cost n_i on any processor p_j is randomly selected from the range-

  $avg\_comp \times (1 - \beta/2) \leq avg\_comp \leq avg\_comp \times (1 + \beta/2)$

Input parameters were assigned the following values in our simulation study.

- n = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
- *CCR* = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0}
- α = {0.5, 1.0, 2.0}
- *out_degree* = {1, 2, 5, 100}
- β = {0.1, 0.5, 0.75, 1.0}

These combinations yield 8640 different DAG types. For each DAG type, 25 different random graphs were generated with the same parameters but with different edge and node weights. Thus a sum total of 216,000 random DAGs were used in the study. The

number of processors was fixed at 10. The processor speeds were randomly selected based on the β value.

Figure 3.45 provides the SLR of HCNF, HEFT and the STDS algorithms for graphs with different Node sizes. Each data point is averaged over 864 distinct readings. The average SLR improvement of HEFT over STDS is 6%, and over HEFT is 10% approximately. Figure 3.46 gives the average speedup versus number of nodes. Each data point is averaged over 864 different readings. The Average improvement in the speedup of the HCNF over STDS is 9%, and over HEFT is 14%. Figure 3.47 provides the average SLR values for CCR values ranging from 0.1 to 1.0 in steps of 0.1. Each data point is averaged over 480 different readings. The average improvement of HCNF over HEFT is 11% and over HEFT is 4 %. Figure 3.48 provides the average SLR values for CCR values ranging from 1.0 to 5.0 in steps of 0.5. Each data point is averaged over 480 different readings. The average improvement of HCNF over STDS is 7% and over HEFT is 11%. For higher CCRs the STDS algorithm performs better than the HEFT algorithm since there is more scope for task duplication. Figure 3.49 provides the average Speedup values for CCR values ranging from 0.1 to 1.0 in steps of 0.1. Each data point is averaged over 480 different readings. The average improvement of HCNF over HEFT is 18% and over HEFT is 9%. Figure 3.50 provides the average SLR values for CCR values ranging from 1.0 to 5.0 in steps of 0.5. Each data point is averaged over 480 different readings. The average improvement of HCNF over STDS is 9% and over HEFT is 5%.

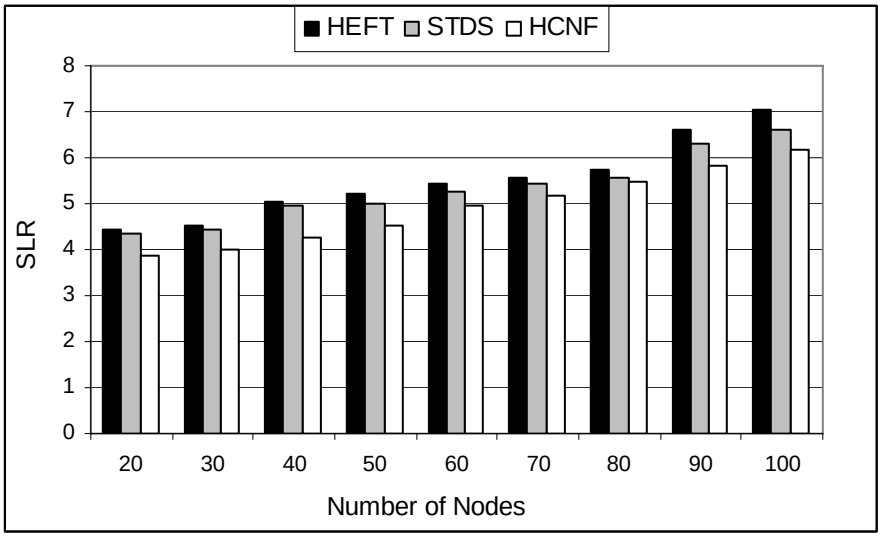

Figure 3.45 Parametric random graphs - SLR vs. number of nodes

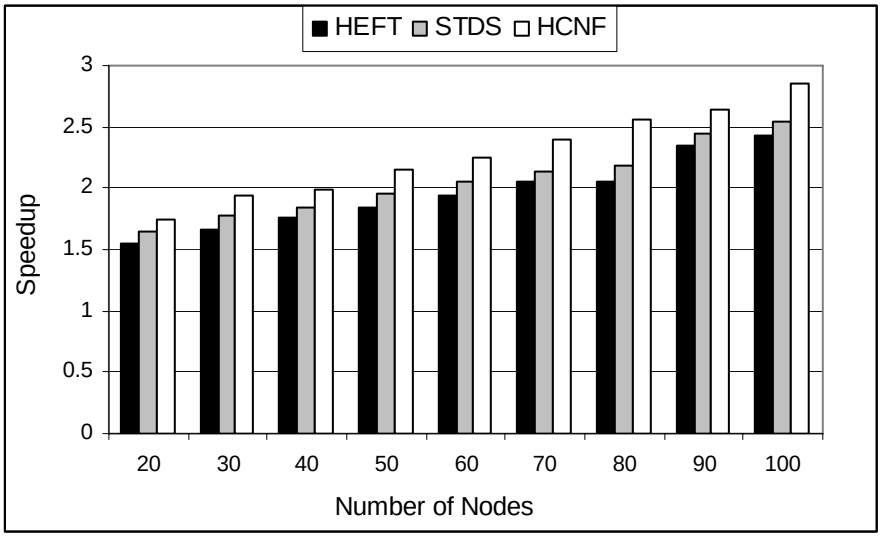

Figure 3.46 Parametric random graphs - Speedup vs. number of nodes

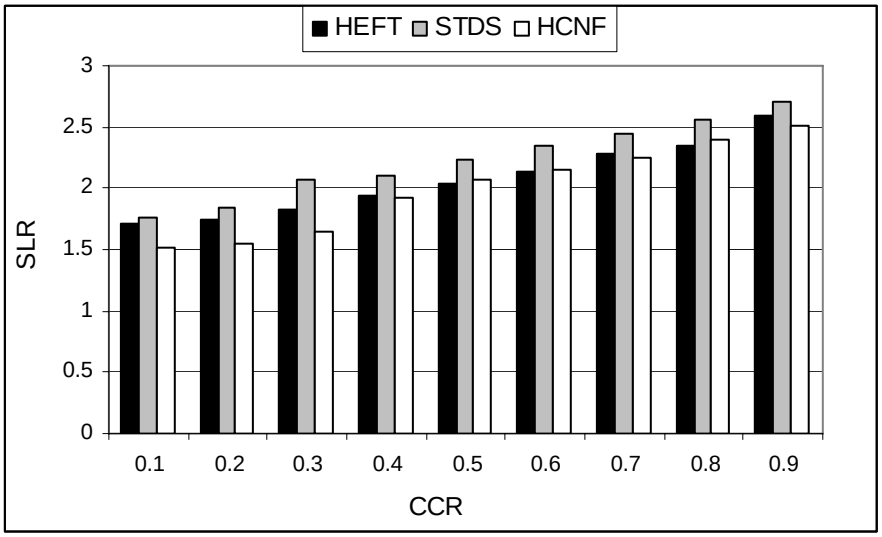

Figure 3.47 Parametric random graphs-SLR vs. CCR (0.1 to 0.9)

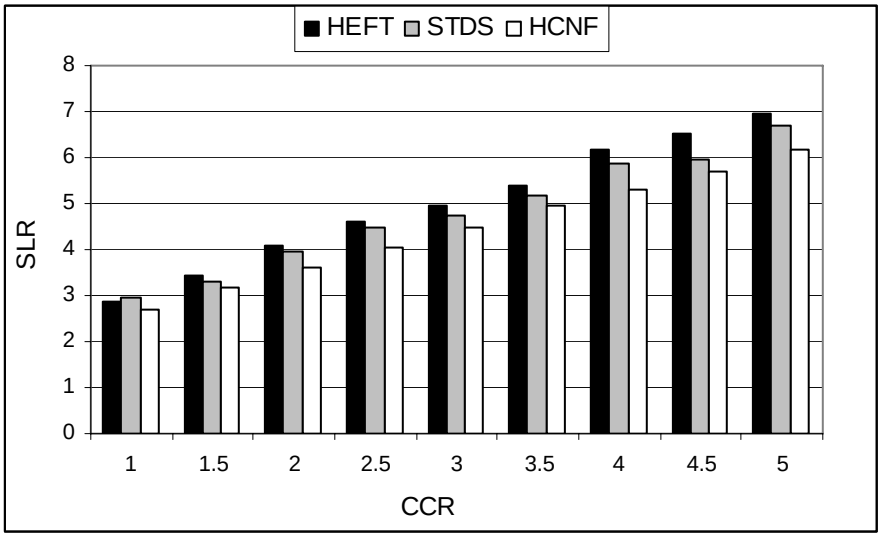

Figure 3.48 Parametric random graphs-SLR vs. CCR (1.0 to 5.0)

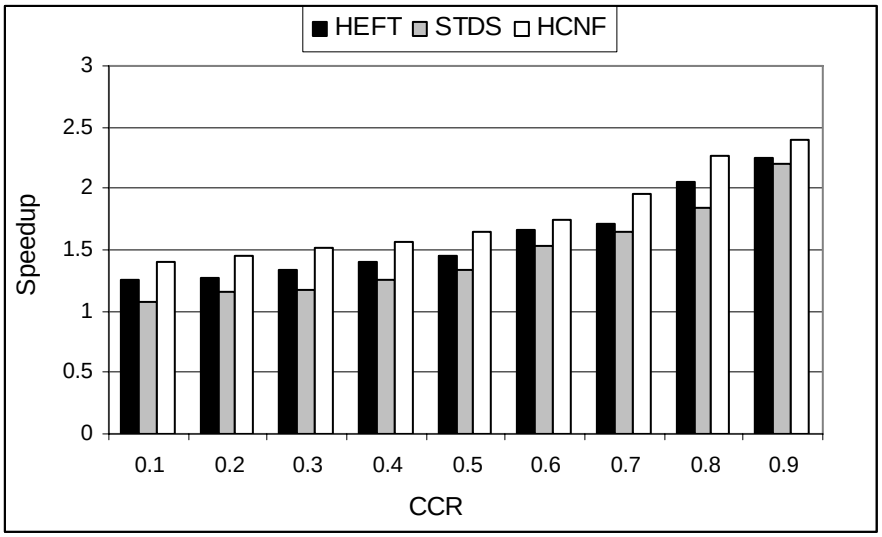

Figure 3.49 Parametric random graphs-Speedup vs. CCR (0.1 to 0.9)

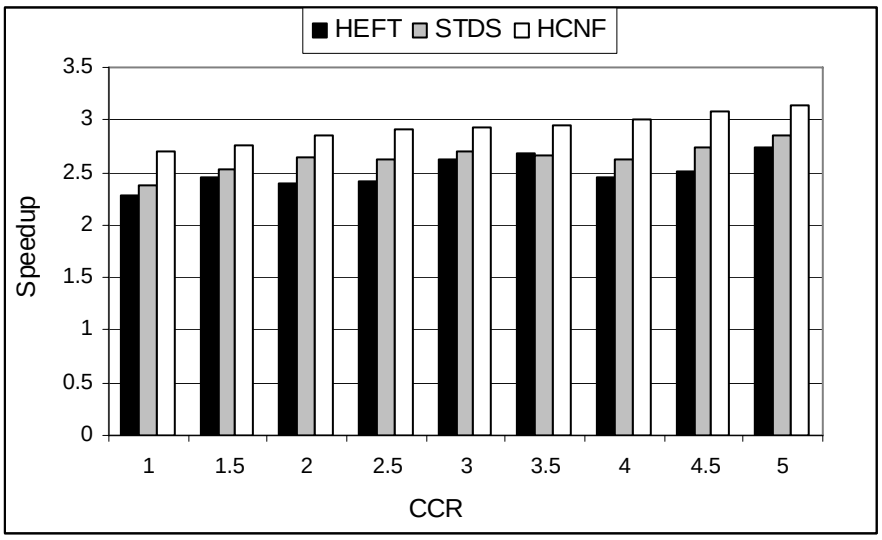

Figure 3.50 Parametric random graphs-Speedup vs. CCR (1.0 to 5.0)

3.5 Conclusion

A new task-duplication based static scheduling algorithm called the Heterogeneous Critical Node First (HCNF) for scheduling DAGs onto a network of heterogeneous processors was proposed. The performance of HCNF, HEFT and STDS was compared using randomly generated graphs, benchmark graphs and parametric graphs. HCNF clearly outperformed both HEFT and STDS with respect to speedup and SLR. The superior performance of HCNF can be attributed to the low-cost task duplication strategy that facilitates earlier start times for many nodes which otherwise have to wait for all the data items to arrive from their favorite predecessors. HCNF can be improved by exploring the possibility of duplicating the second and the third favorite predecessors (if any) to further expedite the start times of nodes. The feasibility of such an approach needs to be investigated.

CHAPTER 4

THE HETEROGENEOUS LARGEST TASK FIRST (HLTF) ALGORITHM

This chapter presents a new low-complexity algorithm called the Heterogeneous Largest Task First (HLTF) for scheduling independent tasks of a meta-task onto a network of heterogeneous processors to minimize the overall execution time. The problem was formally defined in section 2.8. This chapter is organized as follows. Section 4.2 discusses the motivation behind the development of HLTF. Section 4.2 describes the algorithm in detail. Section 4.3 provides the running trace of HLTF. Section 4.4 discusses the theoretical non-equivalence of HLTF and the Sufferage algorithm [23] and section 4.5 provides the simulation study.

4.1 Motivation

A meta-task is a set of independent tasks without any precedence constraints. Scheduling a meta-task onto a set of heterogeneous processors to minimize the overall execution time is a NP-complete problem. Among the scheduling algorithms discussed in the literature review, the Sufferage has the best performance in terms of minimizing the makespan [23]. The time complexity of Sufferage is $O(s^2 * m)$, where s is the size of the meta-task and m the number of processors.

Meta-computing systems such as clusters and grids need to schedule tens of thousands of tasks on a regular basis. A meta-task could contain over a 1000 independent tasks in practical scenarios [23]. Figure 4.1 summarizes the running times of Sufferage. Sufferage takes more than 100 seconds to schedule a meta-task of 1000 tasks. The running time increases with the size of the meta-task. The algorithm takes more than 3000 seconds to schedule a meta-task of 5000 tasks. This can be mainly attributed to the high time complexity $O(s^2 * m)$ of the Sufferage algorithm . The high running times of Sufferage could be a major bottleneck in the scheduling process and could negatively impact the overall performance of a meta-computing system.

To counter this problem, we propose a new low-complexity algorithm called the Heterogeneous Largest Task First (HLTF) to map a meta-task onto a set of heterogeneous processors with an objective to minimize its makespan. Simulation results in chapter 4.5 reveal that in terms of minimizing the makespan, HLTF is at par with Sufferage. However, with respect to running times, HLTF with a lower time complexity of $O(s(\log s + m))$, significantly outperforms Sufferage.

4.2 The Heterogeneous Largest Task First (HLTF) Algorithm

HLTF adapts a simple approach to reduce the overall time complexity of the scheduling process. We first recap the working of the Sufferage algorithm and then explain the working of HLTF. Table 4.1 provides the definition of terms used in HLTF and the Sufferage algorithms.

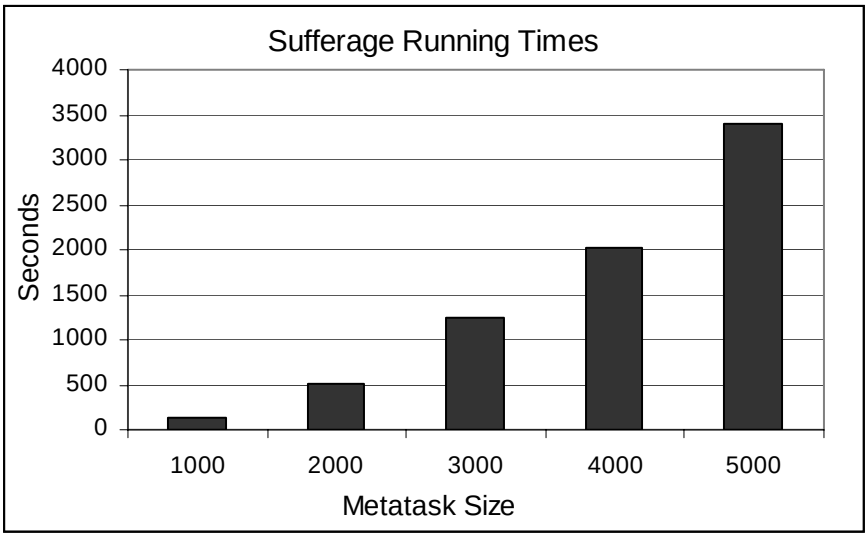

Figure 4.1 Running times of the Sufferage Algorithm

The Sufferage Algorithm

The algorithm is listed in Figure 4.1. At each scheduling step, the Sufferage algorithm picks an arbitrary task from the meta-task set and computes its Earliest Completion Time *(ECT)*, favorite processor *(fproc)* and Sufferage values. If the task's favorite processor has no task previously assigned to it, the current task is tentatively assigned to it. However, if the task's favorite processor has a task already assigned to it, the Sufferages of the current task and the task already assigned are compared. If the Sufferage of the current task is higher, the previously assigned task is removed and the current task is assigned to it. The task that is removed is reinserted into the meta-task. The process is repeated until all the tasks of the Meta-Task set are scheduled.

The HLTF Algorithm

The calculation of Sufferages at each scheduling step, re-inserting the tasks into the meta-task list and repeating all the steps each time a task is reinserted into the list

leads to the high complexity $O(s^2 * m)$ of the Sufferage Algorithm. The HLTF algorithm listed in Figure 4.3 drastically reduces the time complexity of the Sufferage Algorithm by adopting the following approach. Instead of tentatively mapping tasks to processors, HLTF algorithm sorts all tasks in the meta-task set in the non-decreasing order of their sizes before the start of the mapping process. At each scheduling step HLTF picks the largest task in the list and maps it onto a processor that provides its earliest completion time. This seemingly simple approach leads to a very substantial decrease in running time without compromising the performance. Simulation results are reported in Section 4.5. The HLTF algorithm takes $O(s*log\ s)$ to perform merge sort, $O(s* m)$ to compute the completion times of the tasks on all the processors and $O(s * m)$ to compute the earliest completion time of each task. The overall time complexity is $O(s* log\ s+ s*m+ s)$ or $O(s(log\ s)+m))$.

4.3 Theoretical nonequivalence of Sufferage and HLTF algorithms

At each scheduling step, the Sufferage algorithm maps the task with the maximum Sufferage to a machine that provides its earliest finish time. The HLTF algorithm, at each scheduling step, maps the largest task among the candidate tasks to a machine that provide its earliest finish time. Intuitively, the Sufferage and the HLTF algorithms seem to be equivalent.This is because we tend to assume that the largest task will always have the maximum Sufferage i.e for any two tasks t_i and t_j in the meta-task set $(t_1, t_2, t_3, \ldots t_{n-1}, t_n)$ where $t_i > t_j$ and $i < j$, $Sufferage(t_i) > Sufferage(t_j)$. However, this is not the case always and is proved in Theorem 1.

Table 4.1 Definition of Terms used in Sufferage and HLTF

Term	Definition
T	Meta-task set of size s
M	Set of processors available for scheduling
m	Number of processors
$T\_available(p_j)$	Time at which processor p_j can start execution of a new task.
$W_{k,j}$	Running time of task t_k on processor p_j.
$CT(t_k, p_j)$	$T\_available(p_j) + W_{k,j}$ // Execution completion time of task t_k on processor p_j.
$ECT(t_k)$	Min $_{k \in T \& j \in M}\{ CT(t_k, p_j) \}$ //Earliest Completion time of task t_k
$Proc(t_k)$	The processor on which $ECT(t_k)$ can be obtained
$Sufferage(t_k)$	$ECT(t_k)$-Second best $CT(t_k)$

HLTF Algorithm
Sort T using merge sort in non-decreasing order
While $T \neq \Phi$ **do**
    Pick the largest task t_k in T.
        For all $j \in M$
            Compute $CT(t_k, p_j)$
        End For
            Compute $ECT(t_k)$
            $T = T - \{t_k\}$
    Schedule t_k on $Proc(t_k)$
End While
End HLTF

Figure 4.2 HLTF Algorithm

Sufferage Algorithm
While $T \neq \Phi$ **do**
Pick a task $t_k \in T$ in an arbitrary order.
For all $j \in M$
    Compute $CT(t_k, p_j)$
End For
Compute $ECT(t_k)$
$Sufferage(t_k) = ECT(t_k)$-Second best $CT(t_k)$
        If $Proc(t_k)$ has a task t_s already assigned to it
          If $Sufferage(t_k) > Sufferage(t_s)$
            Remove t_s from $Proc(t_k)$ and schedule t_k on $Proc(t_k)$
            $T = T + \{t_s\}$
            $T = T - \{t_k\}$
          End If
        Else
            Schedule t_k on $Proc(t_k)$
            $T = T - \{t_k\}$
        End If
End While
End Sufferage

Figure 4.3 The Sufferage algorithm

Thoerem 1 : For any two tasks t_i and t_j in the meta-task set $(t_1, t_2, t_3, \ldots t_{n-1}, t_n)$ where $t_i > t_j$ and $i < j$, $Sufferage(t_i)$ is not always greater than $Sufferage(t_j)$

Case 1 : Let m_x and m_y be the processors on which tasks t_i and t_j obtain their best *ect* and the next best *ect's* repectively. Let p_x *and* p_y where $p_x > p_y$, be the speeds of processors m_x and m_y in MIPS.

$Sufferage(t_i) = (t_i / p_y + T\_Available(p_y)) - (t_i / p_x + T\_Available(p_x))$

$Sufferage(t_j) = (t_j / p_y + T\_Available(p_y)) - (t_j / p_x + T\_Available(p_x))$

To prove

$Sufferage(t_i) > Sufferage(t_j)$

or, $(t_i/ p_y+T\_Available(p_y)) – (t_i/ p_x + T\_Available(p_x)) > (t_j/ p_y+T\_Available(p_y)) – (t_j/ p_x + T\_Available(p_x))$

or, $t_i/ p_y+T\_Available(p_y) – t_i/ p_x - T\_Available(p_x) > t_j/ p_y+T\_Available(p_y) – t_j/ p_x - T\_Available(p_x)$

or, $t_i/ p_y- t_j/ p_y > t_i/ p_x- t_j/ p_x$ OR

$(t_i- t_j)/ p_y > (t_i- t_j)/ p_x$

Which is true since $t_i > t_j$, $(t_i- t_j) > 0$ and $p_x > p_y$

Case 2 : Let m_x and m_y be the processors on which tasks t_i and t_j obtain their best *ect* and the next best *ect's* repectively. Let p_x *and* p_y where $p_x < p_y$, be the speeds of processors m_x and m_y in MIPS.

$Sufferage(t_i)=(t_i/ p_y+T\_Available(p_y)) – (t_i/ p_x + T\_Available(p_x))$

$Sufferage(t_j)= (t_j/ p_y+T\_Available(p_y)) – (t_j/ p_x + T\_Available(p_x))$

To prove

$Sufferage(t_i) > Sufferage(t_j)$

or, $(t_i/ p_y+T\_Available(p_y)) – (t_i/ p_x + T\_Available(p_x)) > (t_j/ p_y+T\_Available(p_y)) – (t_j/ p_x + T\_Available(p_x))$

or, $t_i/ p_y+T\_Available(p_y) – t_i/ p_x - T\_Available(p_x) > t_j/ p_y+T\_Available(p_y) – t_j/ p_x - T\_Available(p_x)$

or, $t_i/ p_y- t_j/ p_y > t_i/ p_x- t_j/ p_x$ OR

$(t_i- t_j)/ p_y > (t_i- t_j)/ p_x$

Which is NOT true since $t_i > t_j$, $(t_i- t_j) > 0$ and $p_x < p_y$

Therefore for any two tasks t_i and t_j in the meta-task set $(t_1, t_2, t_3,\ldots t_{n-1}, t_n)$ where $t_i > t_j$ and $i < j$, $Sufferage(t_i) > Sufferage(t_j)$ is not always true.

As an example, in Table 4.2 observe that in the third iteration, *T3* is the largest task in the Metatask set and the HLTF algorithm picks *T3* and schedules it onto its favorite processor *p1*. However, notice that the Sufferage of *T3* (14.25) is less than the sufferage of *T2* (16.91), despite *T3* being larger than *T2*. Also, observe that the *fproc1* of all the

tasks is processor 1, the fproc2 of all the tasks is processor 3 and the speed of processor 1 (4 MIPS) is less than that of processor 3 (5 MIPS). This scenario illustrates case 2 of Theorem 1 and provides a practical example of the difference between the Sufferage and the HLTF algorithms

4.4 Simulation and Results

Simulations were conducted on a 440 MHz Sun Ultra 5 machine running on a Solaris 8 Operating System. We compared the relative performance of HLTF and Sufferage w.r.t makespan and running costs. We developed a simulator with the following input parameters.

n : Number of tasks in the metatask.

p: Number of processors in the distributed system.

std_dev: Standard deviation of the metatask

size_min :Minimum task size in MIPS.

size_max: Maximum task size in MIPS.

m: Number of metatasks.

The maximum number of the processors used in our simulations was 20.

4.4.1 Comparison of Makespan

The makespan of various metatasks using HLTF and sufferage was measured using the following input parameters.

n ={50,100,200,300,400,500,750,1000}

P ={5,10,15,20}

std_dev={5,10,15,20,25,30}

size_min ={10}

size_max ={ 100}

m ={1}

The results are shown in Figures 3 to 6. Each data point is an average different readings on 4 different processors. The performance of HLTF was slightly better than that of Sufferage. An important observation was that we did not come across a metatask for which the performance of Sufferage was better than that of HLTF. The Average improvement of HLTF over Sufferage was 0.48%.

4.4.2 Comparison of Running Costs

The running times of Sufferage and HLTF were measured using different metatask sizes. The results are shown in Figures 4.10 to 4.12. Each data point is an average of 25 different readings. The running cost of the Sufferage Algorithm exponentially increases as the size of the met-task increases. For meta-task sizes > 1000, the HLTF provides a very significant reduction in the running costs.

Table 4.2 Example showing nonequivalence of the Sufferage and the HLTF Algorithms:

Sufferage

Meta-Task={*t1,t2,t3,t4,t5,t6*}

Task Sizes t1=157, t2=111, t3=143, t4=128, t5=111, t6=149 (MI)

Processor Speeds p1=4, p2=5, p3=6 (MIPS)

First Iteration

task	*Eft1*	*fproc1*	*eft2*	*fproc2*	*Sufferage*
T1	26.17	3	31.4	2	5.23
T2	18.5	3	22.2	2	3.7
T3	23.83	3	28.6	2	4.77
T4	21.33	3	25.6	2	4.27
T5	18.5	3	22.2	2	3.76
T6	24.83	3	29.8	2	4.97

Schedule t1 on processor 3 Meta-Task={*t2,t3,t4,t5,t6*}

t_avail[1]=0, *t_avail[2]*=0, *t_avail[3]*=26.16

Second Iteration

task	*Eft1*	*fproc1*	*eft2*	*fproc2*	*Sufferage*
T2	22.2	2	27.75	1	5.55
T3	28.6	2	35.75	1	7.15
T4	25.6	2	32	1	6.4
T5	22.2	2	27.75	1	5.55
T6	29.8	2	37.25	1	7.45

Schedule t6 on processor 2 Meta-Task={*t2,t3,t4,t5*}

t_avail[1]=0, *t_avail[2]*=29.8 , *t_avail[3]*=26.16

Third Iteration

task	*Eft1*	*fproc1*	*eft2*	*fproc2*	*sufferage*
T2	27.75	1	44.67	3	16.92
T3	35.75	1	50	3	14.25
T4	32.0	1	47.5	3	15.5
T5	27.75	1	44.67	3	16.92

Schedule t2 on processor 1 Meta-Task={*t3,t4,t5*}

t_avail[1]=27.75, *t_avail[2]*=29.8 , *t_avail[3]*=26.16

HLTF

Meta-Task={*t1,t6,t3,t4,t2,t5*}// Sorted in the non decreasing order using merge sort

First Iteration

Largest task = *t1*

Schedule task *t1* on processor *p3*

t_avail[1]=0, *t_avail[2]*=0, *t_avail[3]*=26.16

Meta-Task={*t6,t3,t4,t2,t5*}

Second Iteration

Largest task = t6

Schedule task *t6* on processor *p2*

t_avail[1]=0, *t_avail[2]*=29.8,

t_avail[3]=26.16

Meta-Task={*t3,t4,t2,t5*}

Third Iteration

Largest task = t3

Schedule task *t3* on processor *p1*

t_avail[1]=35.75, *t_avail[2]*=29.8,

t_avail[3]=26.16

Meta-Task={*t4,t2,t5*}

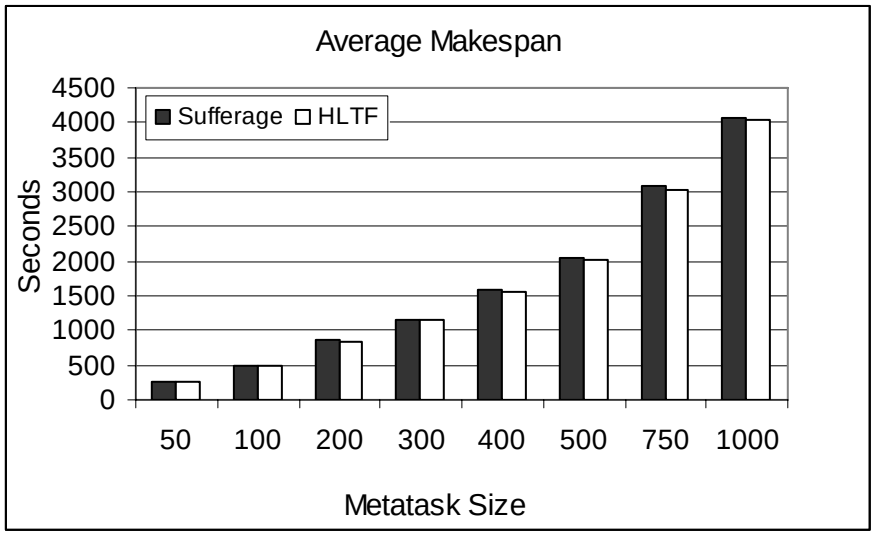

Figure 4.4 Average Makespan of Metatasks, *std_dev*=5

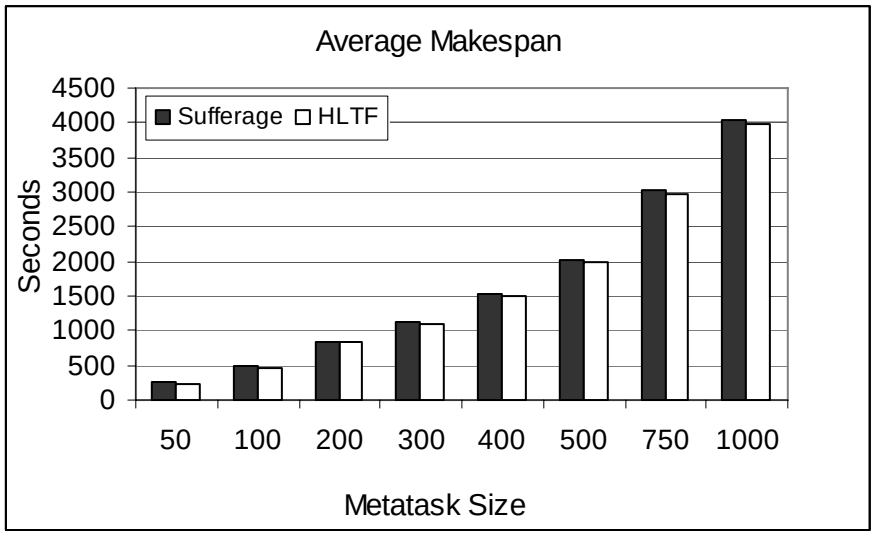

Figure 4.5 Average Makespan of Metatasks, *std_dev*=10

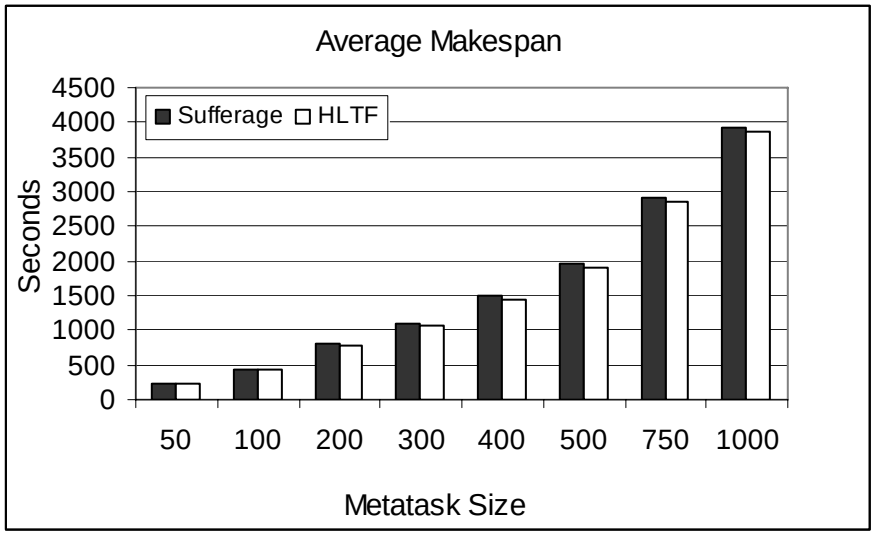

Figure 4.6 Average Makespan of Metatasks, *std_dev* =15

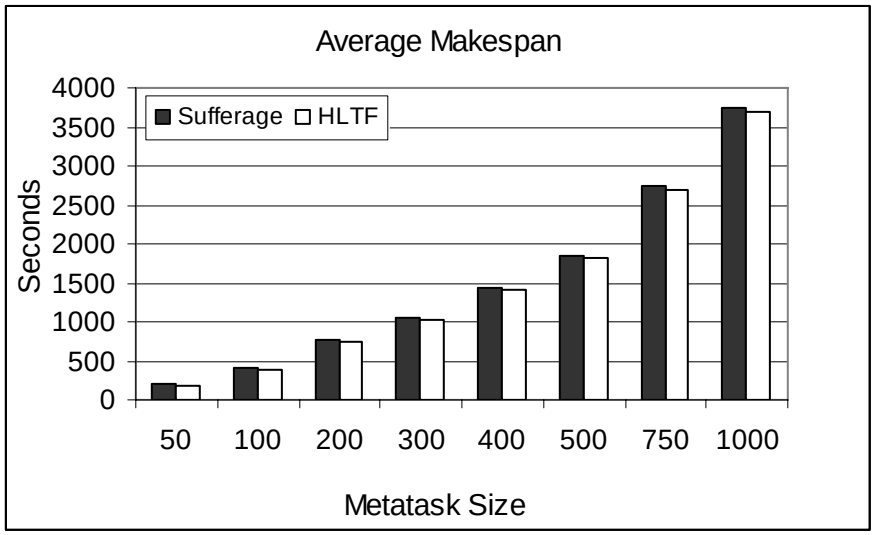

Figure 4.7 Average Makespan of Metatasks, *std_dev*=20

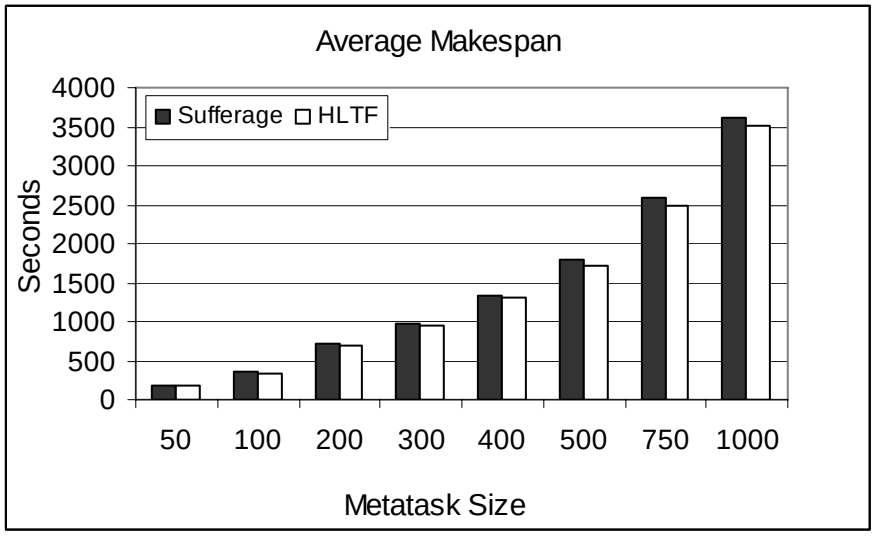

Figure 4.8 Average Makespan of Metatasks *std_dev*=25

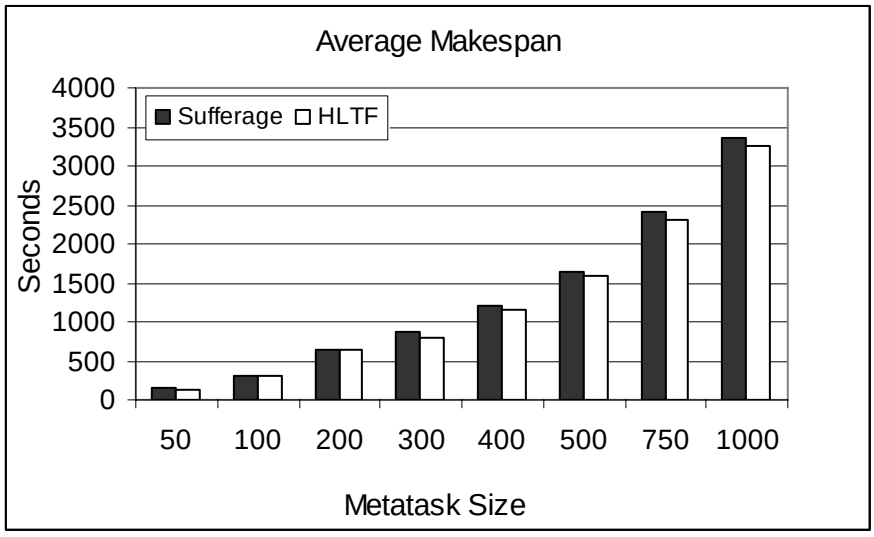

Figure 4.9 Average Makespan of Metatasks, *std_dev*=30

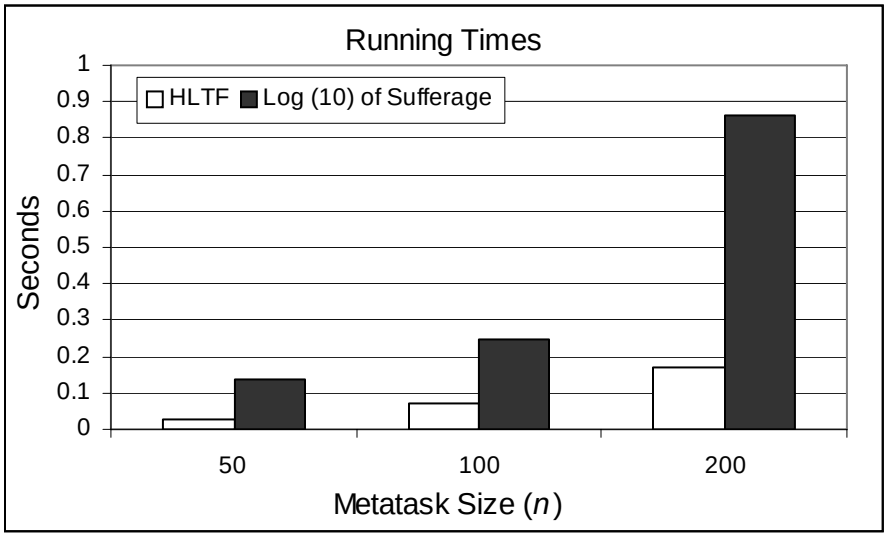

Figure 4.10 Running Times {n =50,100,200}

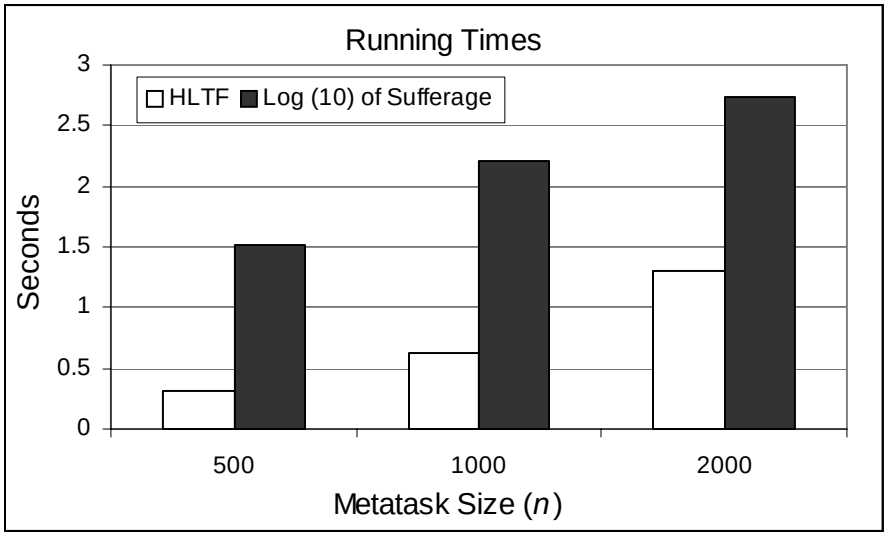

Figure 4.11 Running Times {n =500,1000,2000}

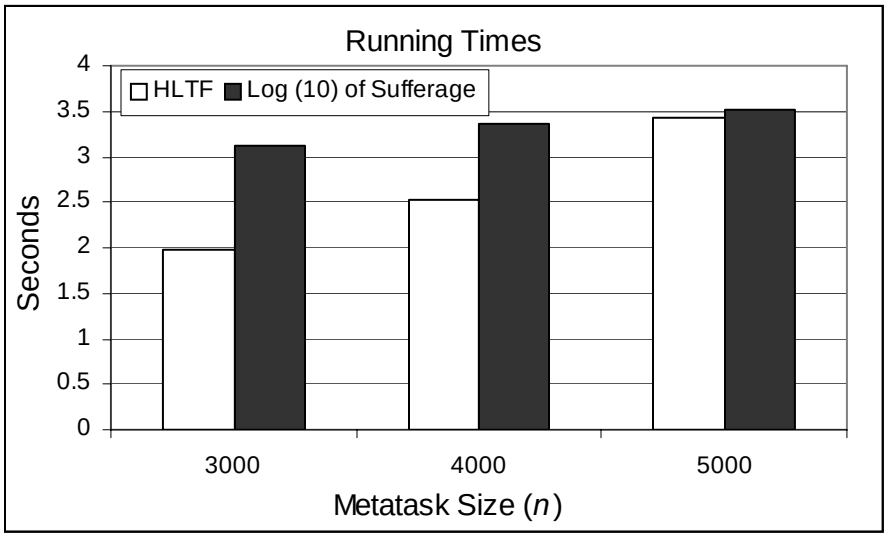

Figure 4.10 Running Times {n =3000,4000,5000}

CHAPTER 5

SCHEDULING INDEPENDENT TASKS WITH DISPATCH TIMES

In this section we introduce a novel heuristic to schedule independent tasks of a meta-task onto a network of heterogeneous processors to minimize the makespan of the meta-task. This section is organized as follows. In Section 5.1 we provide the motivation towards solving this problem. In Section 5.2 we introduce the Earliest Finish Time with Dispatch Time (EFT-DT) algorithm. In Section 5.3 we provide a practical example of the algorithm's working. In Section 5.4 we discuss simulation results.

5.1 Motivation

In meta-computing systems such as the grid, a centralized scheduler may make all scheduling decisions with respect to independent tasks. The scheduler makes a scheduling decision and maps tasks onto processors. In reality, the mapping of tasks onto processors requires time to dispatch the task from the scheduler onto a processor. In the previous works [20] [21][23] related to scheduling independent tasks of a meta-task onto a network of heterogeneous processors, the dispatch times of the tasks have not been considered in making scheduling decisions. The Sufferage, Min-Min and the Min-Max [23] algorithms assume a zero dispatch time in their scheduling model. We believe that in practical scenarios a zero dispatch time is not feasible and may lead to unrealistic schedules. In this section we introduce a novel heuristic to schedule independent tasks of

a meta-task onto a network of heterogeneous processors considering the dispatch times of tasks.

5.2 The Earliest Finish Time with Dispatch Time (EFT-DT) Algorithm

In the EFT-DT algorithm, the priority of a task is defined as the sum of its mean execution time over all the processors and the standard deviation of its execution time over all the processors. At each scheduling step, EFT-DT picks the task with the highest

Table 5.1 EFT-DT Algorithm –Defnition of Terms

Term	Definition
T	Meta-task set of size s
M	Set of processors available for scheduling
m	Number of processors
$mean_k$	Mean execution time of task t_k over all the processors
std_k	Standard deviation of the execution times of task t_k over all the processors
$T\_available(p_j)$	Time at which processor p_j can start execution of a new task.
$W_{k,j}$	Running time of task t_k on processor p_j.
D_{kj}	Time required to dispatch task t_k from the scheduler to processor p_j
$CT(t_k, p_j)$	$= Max\{T\_available(p_j), D_{kj}\} + W_{k,j}$ // Execution completion time of task t_k on processor p_j.
$ECT(t_k)$	$= \text{Min}_{k \in T \,\&\, j \in M}\{ CT(t_k, p_j) \}$ //Earliest Completion time of task t_k
$Proc(t_k)$	The processor on which $ECT(t_k)$ is obtained

priority and schedules it onto a processor that provides its earliest completion time.

EFT-DT Algorithm
For all $t_k \in T$
$priority(t_k) \leftarrow mean_k + std_k$
End For
While $T \neq \Phi$ **do**
Pick a task $t_k \in T$ with the highest priority
  For all $j \in M$
    $CT(t_k, p_j) \leftarrow Max\{T\_available(p_j), D_{kj}\} + W_{k,j}$
  End For
  $ECT(t_k) \leftarrow$ Min $_{k \in T \,\&\, j \in M}\{ CT(t_k, p_j) \}$
  Compute $Proc(t_k)$
  Assign t_k to $Proc(t_k)$
  $T\_available(Proc(t_k)) \leftarrow ECT(t_k)$
  $T = T - \{t_k\}$
End While
End EFT-DT

Figure 5.1 The EFT-DT Algorithm

The completion time of task on a processor is defined as

$CT(t_k, p_j) \leftarrow Max\{T\_available(p_j), D_{kj}\} + W_{k,j}$

to account for the dispatch times. EFT-DT later calculates the processor on which the least completion time is obtained and schedules the task onto it. EFT-DT takes $O(s)$ to compute the priorities of all the tasks and $O(s \times m)$ to calculate the earliest completion times of the tasks. Thus the overall complexity is $O(s \times m)$.

5.3 Example Run of EFT-DT

We now show the working of EFT-DT with a sample meta-task shown in Figure 5.2. Task priorities are computed as follows {8,2,10,1,3,9,4,7,5,6}.

Step1: Schedule task 8 on processor P2

Step 2: Schedule task 2 on processor P3

Table 5.2 A sample metatask

Task	P1	P2	P3
1	15	13	15
2	16	20	16
3	17	16	11
4	20	13	10
5	11	11	12
6	14	14	12
7	15	11	15
8	20	17	13
9	19	10	16
10	18	19	18

Table 5.3 Meta-task Dispatch Times

Task	P1	P2	P3
1	6	6	7
2	9	8	5
3	7	8	9
4	6	9	10
5	8	7	7
6	10	7	6
7	9	7	7
8	6	10	10
9	10	9	5
10	9	8	10

Step 3: Schedule task 10 on processor P1

Step 4: Schedule task 1 on processor P2

Step 5: Schedule task 3 on processor P1

Step 6: Schedule task 9 on processor P3

Step 7: Schedule task 4 on processor P3

Step 8: Schedule task 7 on processor P2

Step 9: Schedule task 5 on processor P2

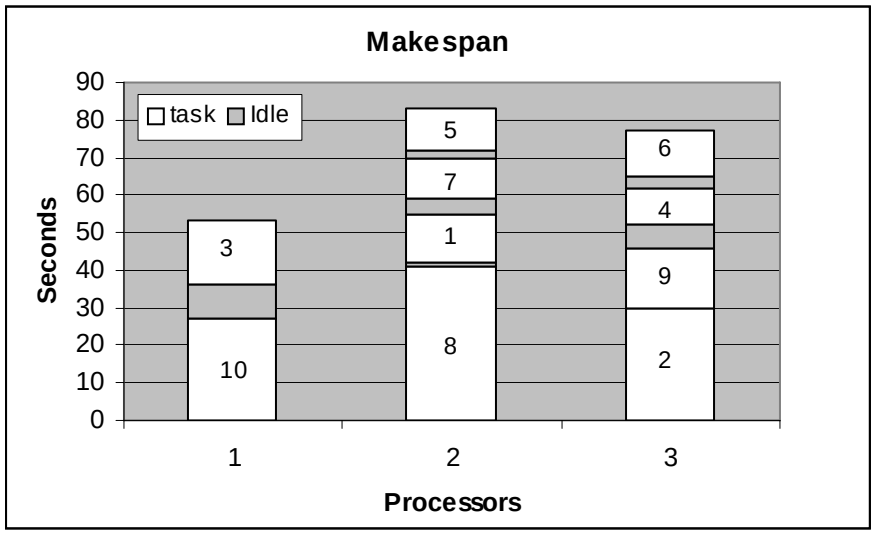

Figure 5.2 Gantt Chart for the Meta-Task

Step 10 : Schedule task 6 on processor P3

The Gantt chart for the meta-task is provide in Figure 5.3

5.4 Simulation Study

We developed a simulator with the following input parameters to compare the performance of EFT-DT and the FIFO approach.

n: Meta-task size

size_max: Maximum size of a task within a meta-task

dis_max: Maximum dispatch time of each task

std_dev: Standard deviation of the meta-task

proc_dev: Standard deviation of the processor speeds

num_proc: Number of processors used.

The input parameters were set with the following values in our simulation study.

n: {1000,2000,5000,7500,10000}

size_max: ={100}

dis_max: {50}

std_dev: {5,10,15,20,25,30}

proc_dev: {2,4,6}

num_proc: {5,10,15,20}

Table 5.4 Parameter Values

Parameter	Minimum	Maximum	Standard Deviation
Task Size	10	100	5-30
Dispatch Times	10	50	X
Proc Speeds	1	10	2,4,6
No of tasks	1000	10000	X
No of Processors	5	20	X

Each data point in the graphs that follow is an average of 4 different readings obtained using different number of processors. Figure 5.3 compares the makespan of EFT-DT and

FIFO for *std_dev*=5 and *proc_dev*=2. The average improvement of EFT-DT is 28%. Figure 5.4 provides the comparison for *std_dev*=10 and *proc_dev*=2. The average improvement of EFT-DT is 29%. Figure 5.5 provides the comparison for *std_dev*=15 and *proc_dev*=2. The average improvement of EFT-DT is 28%. Figure 5.6 provides the comparison for *std_dev*=20 and *proc_dev*=2.The average improvement of EFT-DT is 30%. Figure 5.7 provides the comparison for *std_dev*=25 and *proc_dev*=2. The average improvement of EFT-DT is 29%. Figure 5.8 provides the comparison for *std_dev*=30 and *proc_dev*=2. The average improvement of EFT-DT was 30%. Figure 5.9 provides the comparison for *std_dev*=5 and *proc_dev*=4. The average improvement of EFT-DT was 29%. Figure 5.10 provides the comparison for *std_dev*=10 and *proc_dev*=4. The average improvement of EFT-DT was 28%. Figure 5.11 provides the comparison for *std_dev*=15 and *proc_dev*=4. The average improvement of EFT-DT was 31%. Figure 5.12 provides the comparison for *std_dev*=20 and *proc_dev*=4. The average improvement of EFT-DT was 31%. Figure 5.13 provides the comparison for *std_dev*=25 and *proc_dev*=4. The average improvement of EFT-DT is 30%. Figure 5.14 provides the comparison for *std_dev*=30 and *proc_dev*=4. The average improvement of EFT-DT is 29%. Figure 5.15 provides the comparison for *std_dev*=5 and *proc_dev*=6. The average improvement of EFT-DT is 30%. Figure 5.16 provides the comparison for std_dev=10 and proc_dev=6. The average improvement of EFT-DT is 29%. Figure 5.17 provides the comparison for std_dev=15 and proc_dev=6. The average improvement of EFT-DT is 32%. Figure 5.18 provides the comparison for std_dev=20 and proc_dev=6. The average improvement of EFT-DT is 28%. Figure 5.19 provides the comparison for std_dev=25 and proc_dev=6.The average improvement of EFT-DT is 32%. Figure 5.20 provides the

comparison for std_dev =30 and proc_dev=6.The average improvement of EFT-DT is 30%. From all these average improvements, the overall average improvement of EFT-DT over FIFO is 29%

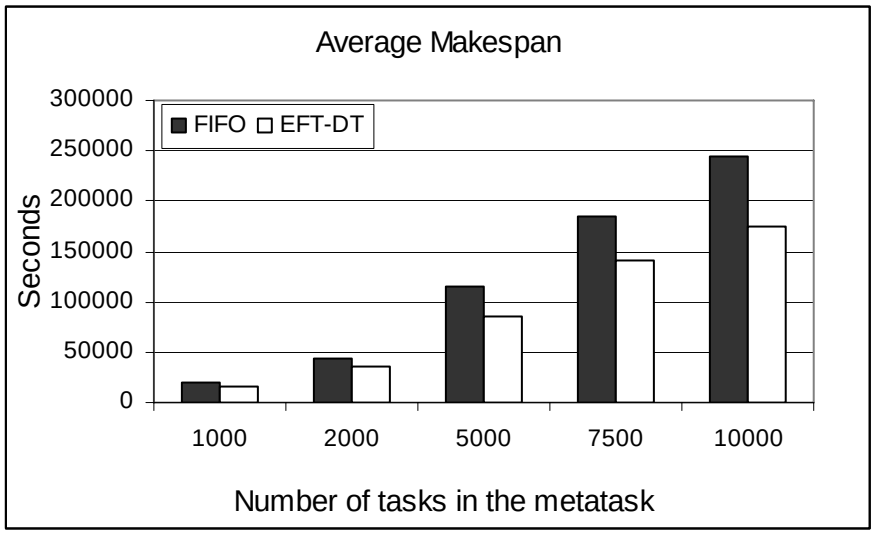

Figure 5.3 Average Makespan- *std_dev*=5, *proc_dev*=2

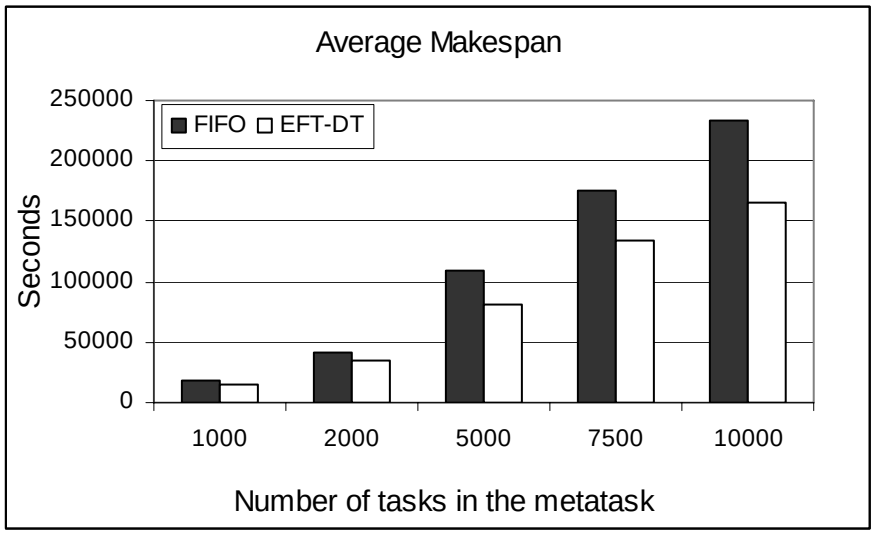

Figure 5.4 Average Makespan- *std_dev*=10, *proc_dev*=2

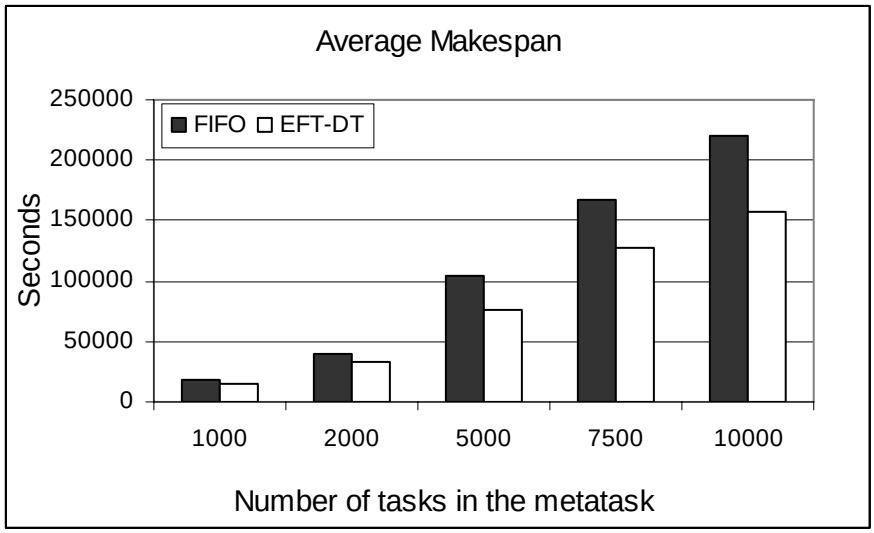

Figure 5.5 Average Makespan- *std_dev*=15, *proc_dev*=2

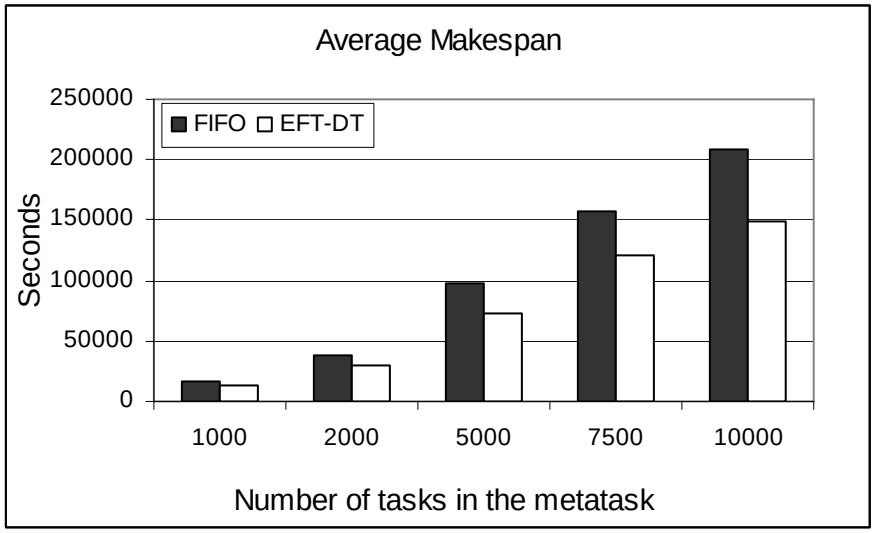

Figure 5.6 Average Makespan- *std_dev*=20, *proc_dev*=2

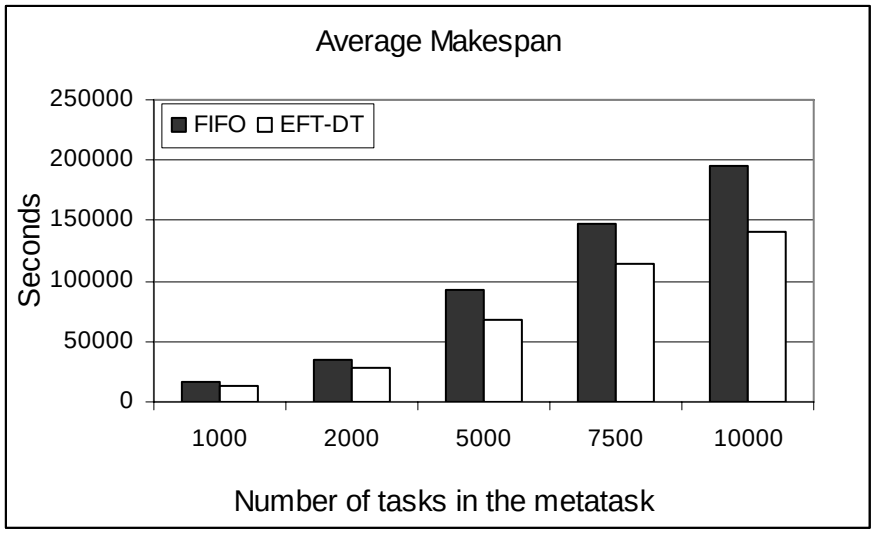

Figure 5.7 Average Makespan- *std_dev*=25, *proc_dev*=2

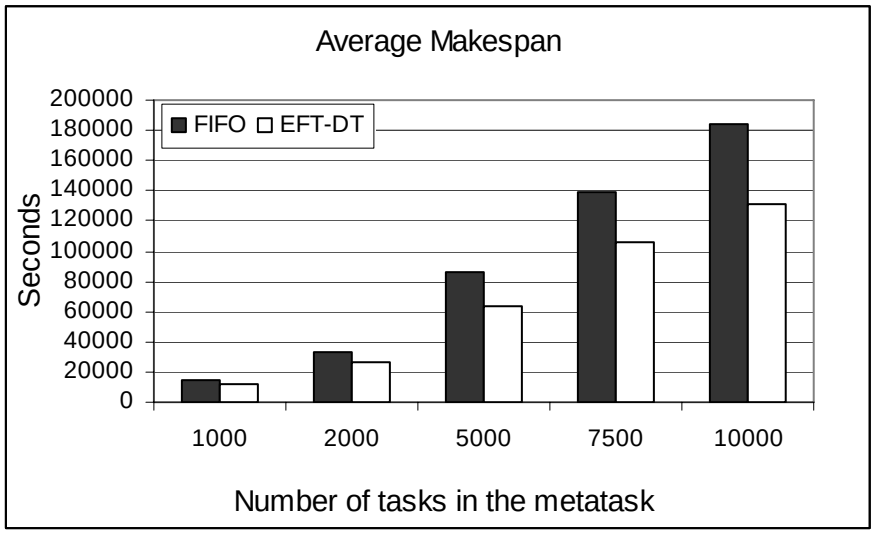

Figure 5.8 Average Makespan- *std_dev*=30, *proc_dev*=2

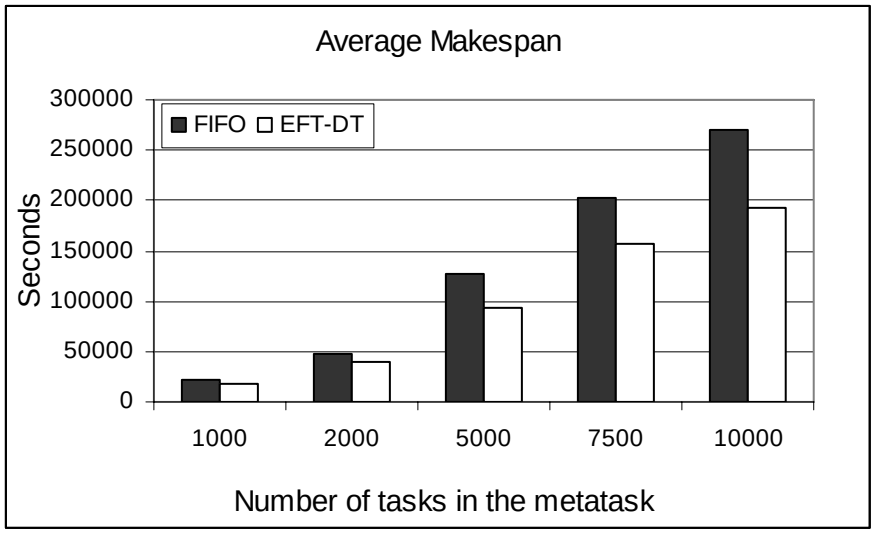

Figure 5.9 Average Makespan- *std_dev*=5, *proc_dev*=4

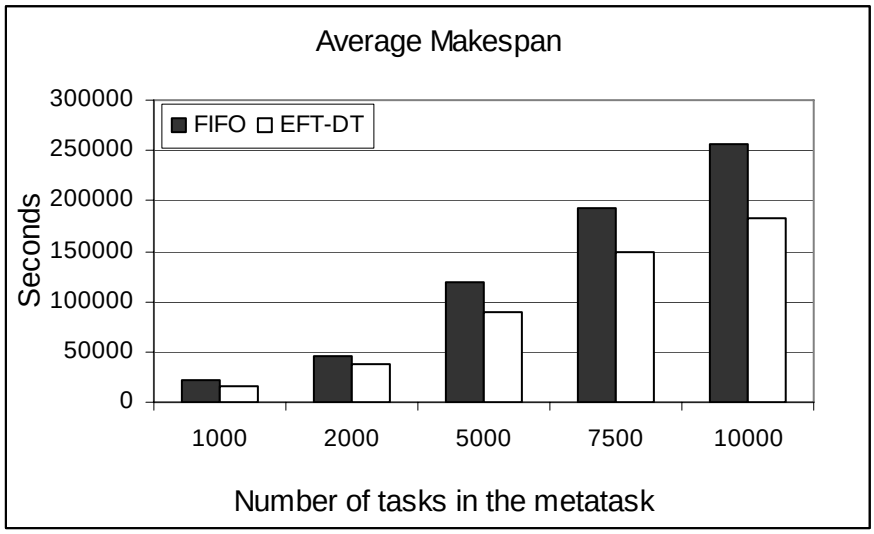

Figure 5.10 Average Makespan- *std_dev*=10, *proc_dev*=4

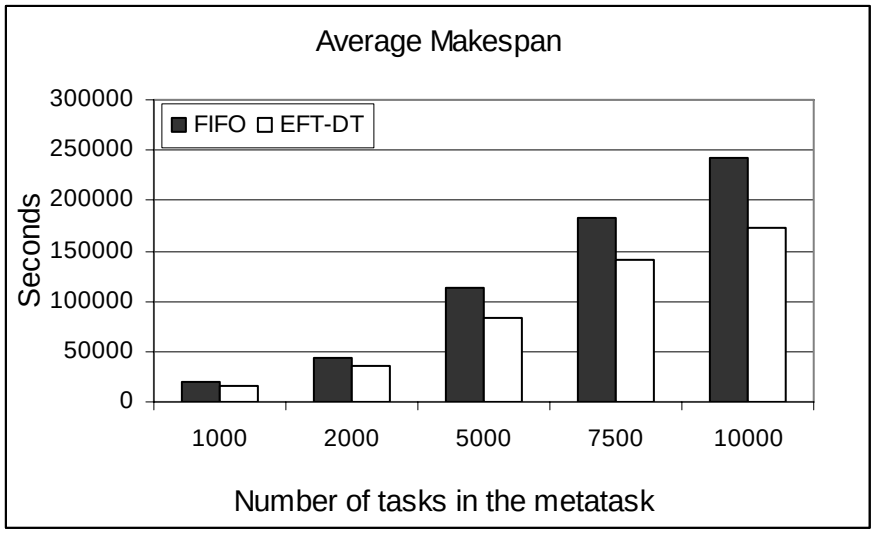

Figure 5.11 Average Makespan- *std_dev*=15, *proc_dev*=4

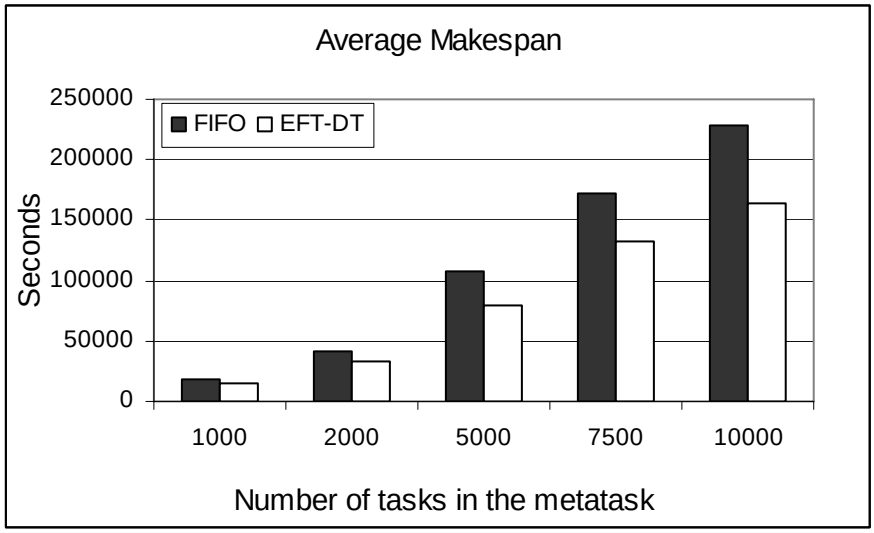

Figure 5.12 Average Makespan- *std_dev*=20, *proc_dev*=4

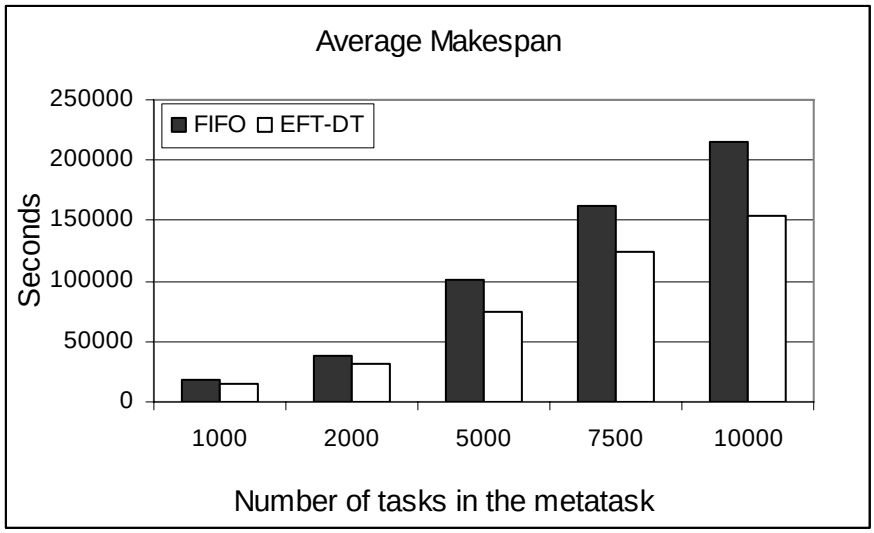

Figure 5.13 Average Makespan- *std_dev*=25, *proc_dev*=4

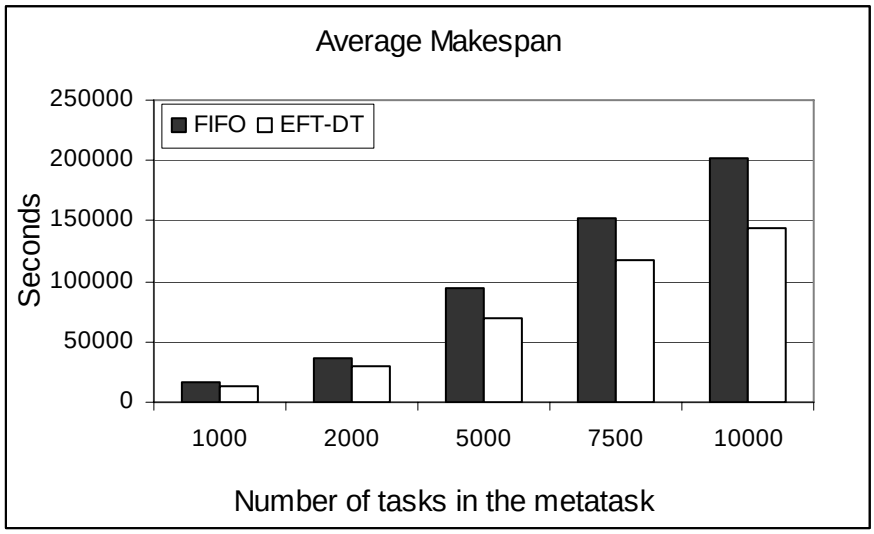

Figure 5.14 Average Makespan- *std_dev*=30, *proc_dev*=4

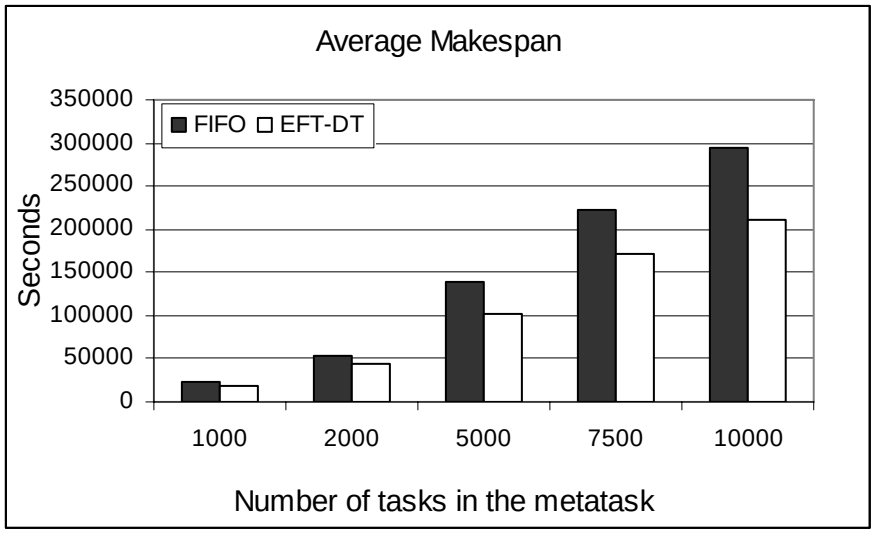

Figure 5.15 Average Makespan- *std_dev*=5, *proc_dev*=6

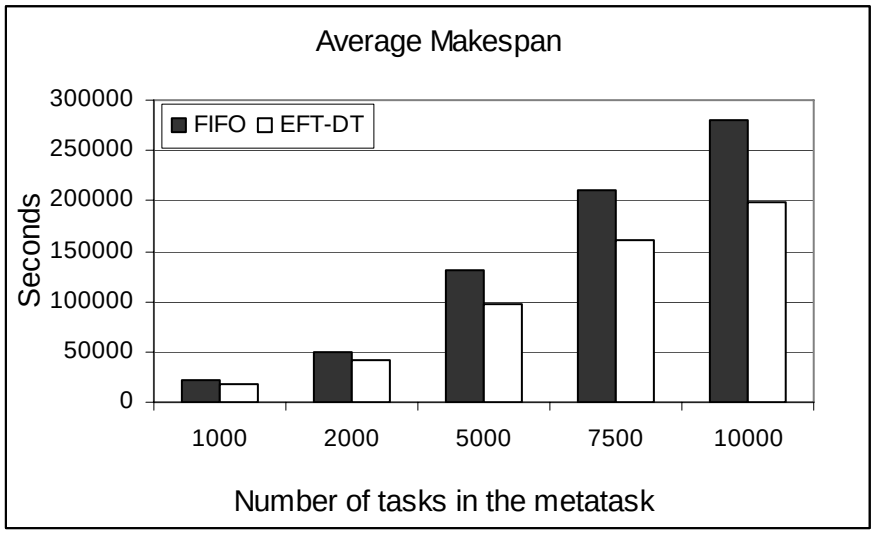

Figure 5.16 Average Makespan- *std_dev*=10, *proc_dev*=6

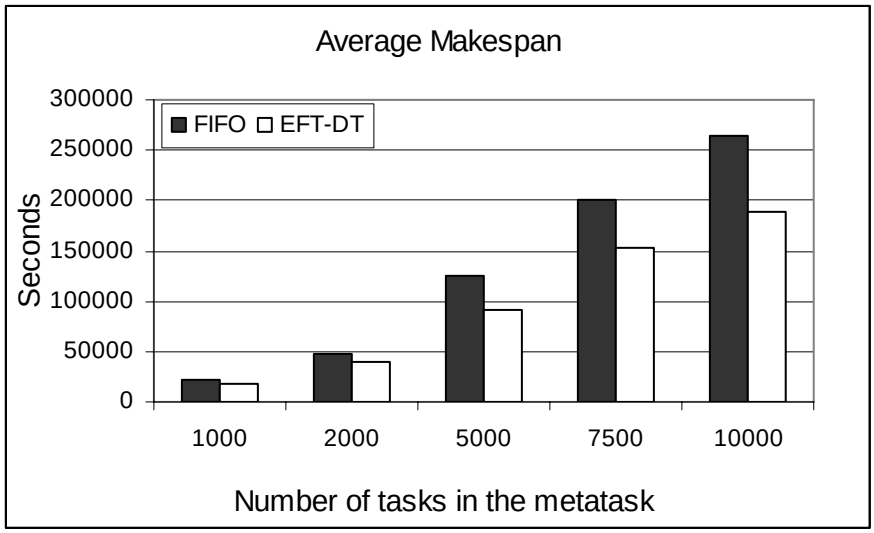

Figure 5.17 Average Makespan- *std_dev*=15, *proc_dev*=6

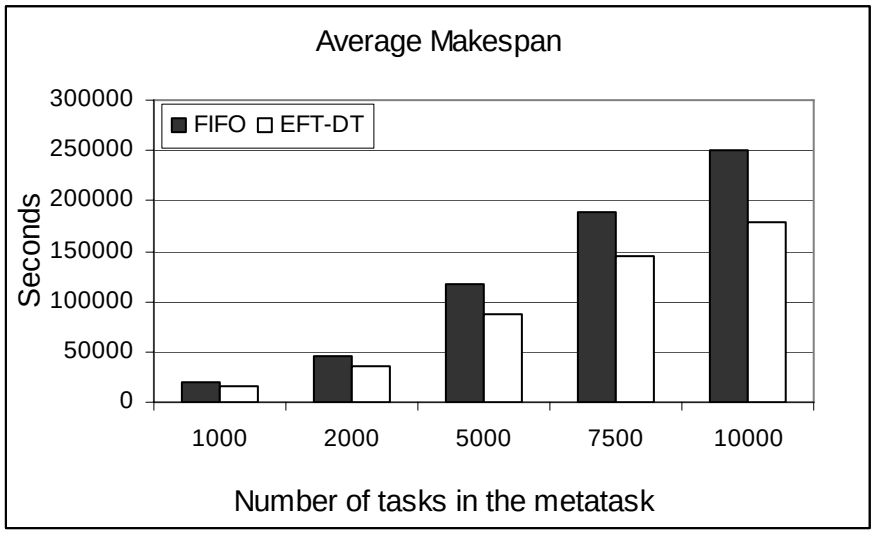

Figure 5.18 Average Makespan- *std_dev*=20, *proc_dev*=6

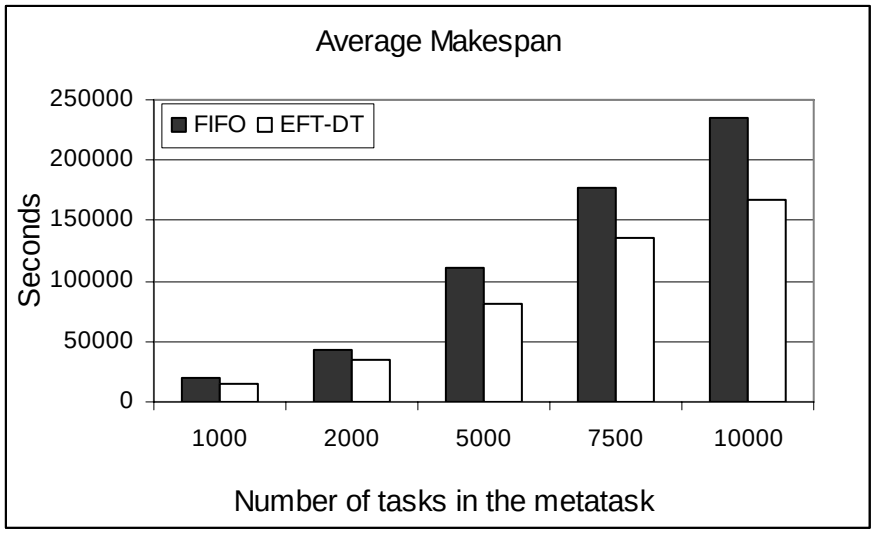

Figure 5.19 Average Makespan- *std_dev*=25, *proc_dev*=6

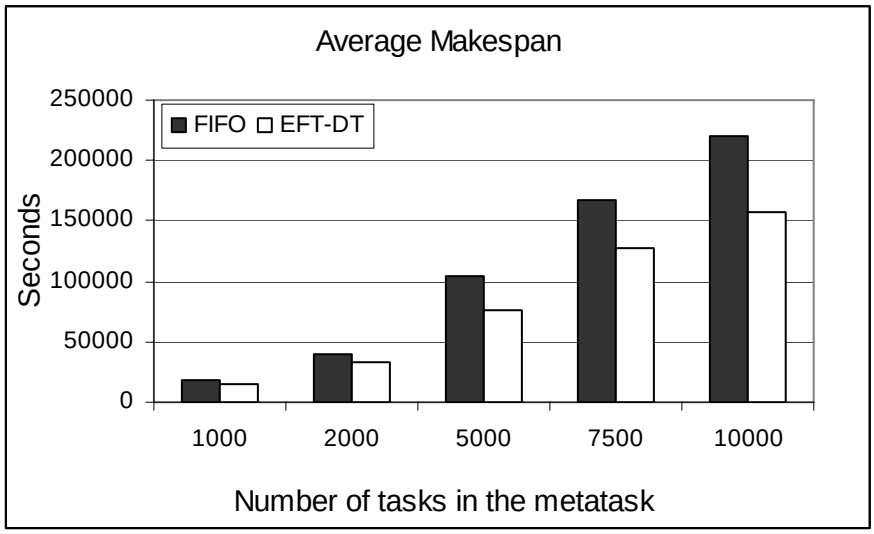

Figure 5.20 Average Makespan- *std_dev*=30, *proc_dev*=6

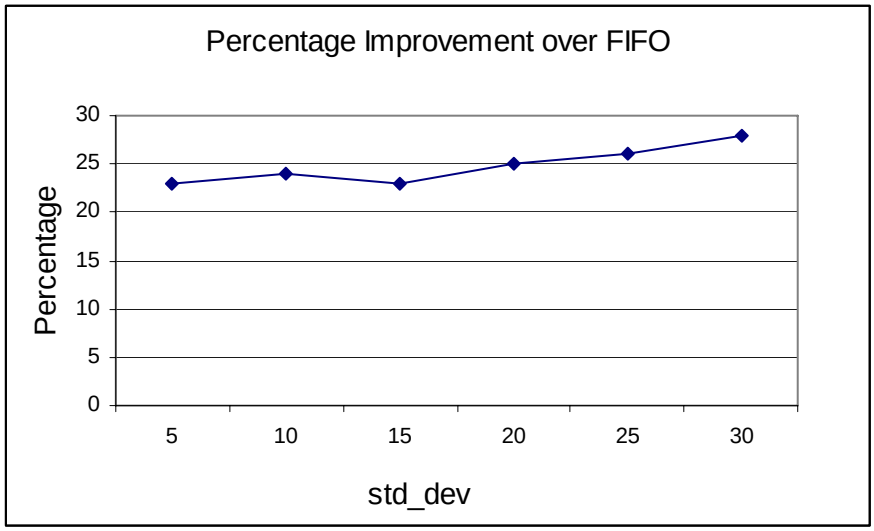

Figure 5.21 Percentage improvement of EFT-DT over FIFO for various *std_dev*

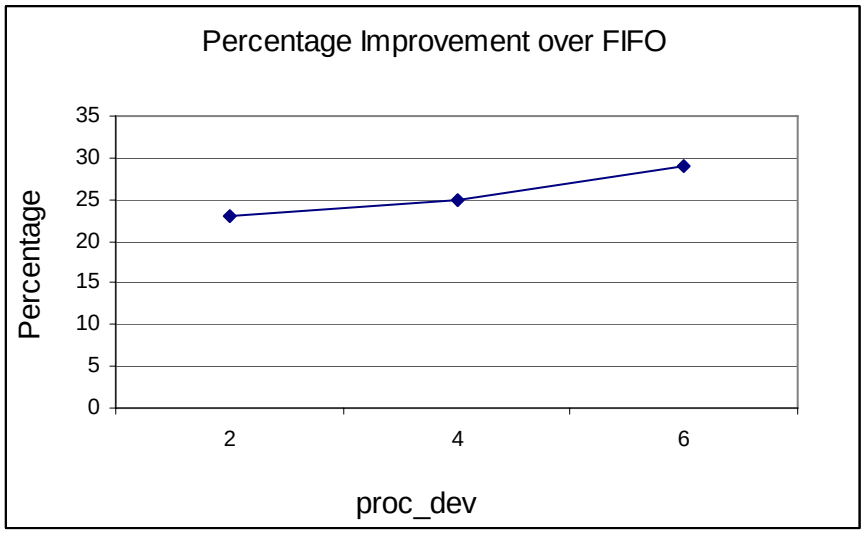

Figure 5.22 Percentage improvement of EFT-DT over FIFO for various *proc_dev*

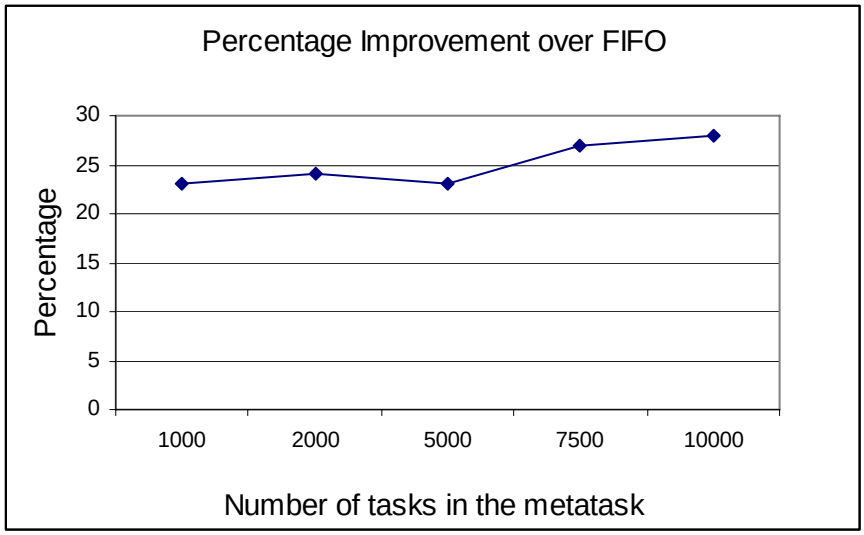

Figure 5.23 Percentage improvement of EFT-DT over FIFO for various metatask sizes

CHAPTER 6

CONCLUSION

In this dissertation, we presented three new algorithms: the Heterogeneous Critical Node First (HCNF) algorithm; the Heterogeneous Largest Task First (HLTF) algorithm and the Earliest Finish Time with Dispatch Time (EFT-DT) algorithm. These algorithms were compared against existing algorithms through extensive simulation. The simulation results were presented in the earlier chapters. In this chapter, we would like to summarize the simulation results briefly and provide concluding remarks.

With respect to the HCNF algorithm, the experimental test suite consisted of application graphs, trace graphs, RGBOS, RGPOS graphs etc. HCNF was compared against HEFT using these graphs and was later compared using both HEFT and STDS using the parametric random graph generator. We now summarize the experimental results.

The SLR and speedup of HCNF and HEFT was compared using graphs of different sizes. The average SLR of HCNF was better than HEFT by 12.3% and the speedup was better than HEFT by 7.9 %. The SLR and Efficiency of HEFT and HCNF were compared using DAGs representing the Gaussian Elimination algorithm. HCNF outperformed HEFT by an average of 25.7% with respect to SLR. With respect to efficiency, HCNF outperformed HEFT by an average of 22.6%. The SLR and speedup of

HEFT and HCNF was compared using trace graphs. HCNF outperformed HEFT in SLR and speedup by an average of 29.5% and 38.4% respectively. The SLR and speedup of HEFT and HCNF was compared using RGBOS graphs whose optimal schedule can be obtained using branch and bound technique. HNCF outperformed HEFT in SLR and speedup by 32.5% and 24.6% respectively. The SLR and speedup of HEFT and HCNF was compared using the RGPOS graphs. HCNF outperformed HEFT in SLR and speedup by 21.1% and 16.9% respectively. The SLR and speedup of HEFT and HCNF was compared using application graphs.These graphs represent a few numerical parallel application programs. This set contains of over 320 graphs in six categories: Cholesky factorization, LU decomposition, Gaussian elimination, FFT, Laplace transforms and Mean Value Analysis (MVA). The number of nodes ranges from 100 to 300. On an average, HCNF outperformed HEFT in SLR and Speedup by 27.5% and 22.7% respectively. The SLR of HCNF, HEFT and the STDS algorithms was compared using the parametric random graph generator. The average SLR improvement of HEFT over STDS is 6%, and over HEFT is 10% approximately. The speedup of HCNF, HEFT and the STDS algorithms was compared using a parametric random graph generator. The Average improvement in the speedup of the HCNF over STDS is 9%, and over HEFT is 14%. The average SLR values for CCR values ranging from 0.1 to 1.0 in steps of 0.1 was compared using the parametric random graph generator. The average improvement of HCNF over HEFT is 11% and over HEFT is 4 %. The average SLR values for CCR values ranging from 1.0 to 5.0 in steps of 0.5 was compared using the parametric random graph generator. The average improvement of HCNF over STDS is 7% and over HEFT is 11%. The superior performance of HNCF can be attributed to the low-cost task

duplication strategy that facilitates earlier start times for many nodes which otherwise have to wait for all the data items to arrive from their favorite predecessors. HCNF can be improved by exploring the possibility of duplicating the second and the third favorite predecessors (if any) to further expedite the start times nodes. The feasibility of such an approach needs to be investigated.

The average makespan of HLTF and the Sufferage algorithms was compared using different metatask sizes and different *std_dev*. The average improvement of HLTF over Sufferage was 4.13%.

The running times of HLTF and Sufferage were compared using metatasks of different sizes. For metatask sizes greater than 1000, HLTF shows an improvement of over a 1000%. The superior performance of HLTF in terms of running times can be attributed to the low complexity sorting technique that is used by the algorithm.

Experiments were conducted to compare the makespan of EFT-DT and FIFO. The overall average improvement of EFT-DT over FIFO is 30%. The superior performance of EFT-DT over FIFO can be attributed to the dispatch times and task execution times occurring in parallel.

BIBLIOGRAPHY

[1] A. Abraham, R. Buyya and B. Nath, "Nature's Heuristics for Scheduling Jobs on Computational Grids," *Proc. ADCOM 2000*, pp. 45 - 52, Cochin India.

[2] V. A. F. Almeida , I. M. M. Vasconcelos , J. N. C. Rabe and D. A. Menasc, "Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems,"*Proc. 1992 ACM/IEEE conference on Supercomputing*, pp. 683-691, Nov. 1992.

[3] J. R. Allen and K. Kennedy, "PFC: A Program to Convert FORTRAN to Parallel Form," *Proc. of the IBM Conference on Parallel Computers and Scientific Computations*, March 1982.

[4] R. Bajaj and D.P. Agarwal, "Improving Scheduling of Tasks in a Heterogeneous Environment," *IEEE Trans. Parallel and Distributed Systems, Vol. 15 No. 2*,pp. 107-118 February 2004.

[5] S. Baskiyar, "Scheduling DAGs on Message Passing m-Processors Systems," *IEICE Trans. Information and Systems*, v E-83-D, no. 7, Oxford University Press, July 2000.

[6] S. Baskiyar, "Scheduling Task-In Trees on Distributed Memory Systems," *IEICE Trans. Information and Systems*, vol. E-84-D, no. 6, June 2001.

[7] S. Baskiyar and P.C. SaiRanga, "Scheduling DAGs on Heterogeneous Multiprocessor Systems to Minimize Finish Time," *Proc. ISCA PDCS*, Reno, Nevada, Aug 2003.

[8] S. Baskiyar and P.C. SaiRanga, "Scheduling DAGs on Heterogeneous Network of Workstations to Minimize Schedule Length," *Proc. ICPP Workshops*, Taiwan, Oct 2003.

[9] S..Baskiyar and P.C. SaiRanga, "Scheduling independent tasks of a metatask with significant dispatch times," Technical Report # CSSE06-03, Auburn University, Nov 2006.

[10] S.Baskiyar and P.C SaiRanga, "Scheduling DAGs on Heterogeneous Network of Workstations to Minimize Finish Time," *Trans. IJCA*, Vol 13, No 4, Dec 2006.

[11] O. Beaumont, V. Boudet, and Y. Robert, "A Realistic Model and an Efficient Heuristic for Scheduling with Heterogeneous Processors," *Proc. IPDPS*, 2002.

[12] C. Boeres, G. Chochia and P. Thanisch, "On the Scope of Applicability of the ETF Algorithm," *Proc. Workshop on Parallel Algorithms for Irregularly Structured Problems*, pp. 159-164, 1995.

[13] R. Buyya, D. Abramson, and J. Giddy, "An Economy Driven Resource Management Architecture for Global Computational Power Grids," *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, June 26-29, 2000, Las Vegas, USA, CSREA Press, USA, 2000.

[14] R. Buyya, D. Abramson, and J. Giddy, "Nimrod-G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid," *Proc. 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2000)*, May 2000, Beijing, China, IEEE Computer Society Press, USA.

[15] R. Buyya, D. Abramson, and J. Giddy, "A Case for Economy Grid Architecture for Service-Oriented Grid Computing," *Proc. International Parallel and Distributed Processing Symposium: 10th IEEE International Heterogeneous Computing Workshop (HCW 2001)*, April 23, 2001, San Francisco, California, USA, IEEE CS Press, USA, 2001

[16] W.Y Chan and C.K. Li, "Heterogeneous Dominant Sequence Cluster (HDSC): a low complexity heterogeneous scheduling algorithm," *Proc. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, 1997, Vol. 2* , pp. 956-959, Aug. 1997.

[17] W.Y. Chan and C.K. Li, "Scheduling tasks in DAG to heterogeneous processor system," *Proc. Sixth Euromicro Workshop on Parallel and Distributed Processing(PDP '98)*, pp. 27-31, Jan. 1998.

[18] H. B. Chen, B. Shirazi, K. Kavi, and A. R. Hurson, "Static scheduling using linear clustering and task duplication," *Proc. ISCA International Conference on Parallel and Distributed Computing and systems*, 1993, pp. 285-290.

[19] C. Chiang, C. Lee, and M. Chang, "A Dynamic Grouping Scheduling for Heterogeneous Internet-Centric Metacomputing System," *Proc. ICPADS, pp.* 77-82, 2001.

[20] W.Y. Chan and C.K. Li, "Scheduling Tasks in DAG to Heterogeneous Processors System," *Proc. 6th Euromicro Workshop on Parallel and Distributed Processing,* Jan.1998.

[21] M. Chetty and R. Buyya, "Weaving computational grids: how analogous are they with electrical grids,**"** *IEEE Trans. Computational Science and Enginering*, Volume 4, Issue 4, July-Aug. 2002, Pages:61 – 71.

[22] B. Cirou and E. Jeannot, "Triplet : a Clustering Scheduling Algorithm for Heterogeneous Systems," *Proc. IEEE ICPP International Workshop on Metacomputing Systems and Applications (MSA'2001)*, sept. 2001, Valencia, Spain

[23] T.H Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms,* The MIT Press, 1990.

[24] M. Cosnard and E. Jeannot, "Compact DAG representation and it's dynamic Scheduling," *Journal of Parallel and Distributed Computing*, Vol. 58, No. 3, September 1999, pp. 487-514.

[25] S. Darbha and D. P. Agrawal, "A task duplication based scalable scheduling algorithm for distributed memory systems", *Journal of parallel and Distributed Computing*, Vol. 46, No. 1, October 1997, pp. 15-27.

[26] S. Darbha and D.P. Agrawal, "Optimal Scheduling algorithm for distributed memory machines", *IEEE Trans. Parallel and Distributed Systems*, Vol. 9, No. 1, January 1998, pp. 87-95.

[27] A. Dogan and F. Ozguner, "Stochastic Scheduling of a Meta-task in Heterogeneous Distributed Computing," *Proc. ICPP Workshop on Scheduling and Resource Management for Cluster Computing*, 2001.

[28] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, "Scheduling Resources in Multi-User, Heterogeneous, Computing Environments with SmartNet," *Proc. 7th IEEE Heterogeneous Computing Workshop (HCW '98)*, Mar.1998, pp. 184–199.

[29] M.Grajcar, "Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system," *Proc. 36th Design Automation Conference*, pp. 280-285, 1999.

[30] A. Gerasoulis and T. Yang, "A comparison of clustering heuristics for scheduling directed acyclic graphs onto multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 16, No. 4, December 1992, pp. 276-291.

[31] D. Hensgen, M. Maheswaran, S. Ali, and H.J. Siegal, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," *Proc. Heterogeneous Computing Workshop*, 1999.

[32] J. Huang and S.Y.Lee, "Effects of Spatial and Temporal Heterogeneity on Performance of a Target Task in Heterogeneous Computing Environments," *Proc. 15th ISCA International Conference on Parallel and Distributed Systems*, Sept. 2002.

[33] C. C. Hui and S. T. Chanson, "Allocating task interaction graphs to processors in heterogeneous networks," *IEEE Trans. Parallel and Distributed Systems,* Vol. 8, No. 9, September 1997, pp. 908-926.

[34] M. Iverson, F. Ozguner, and G. Follen, "Parallelizing Existing Applications in a Distributed Computing Environment," *Proc. Heterogeneous Computing Workshop*, pp. 93-100, 1995.

[35] M. A. Iverson, F. Ozguner and L.C. Potter, "StatisticalPrediction of Task Execution Times Through Analytic Benchmarking for Scheduling in a Heterogeneous Environment," *Proc. 8th Heterogeneous Computing Workshop (HCW '99)*, p. 99, April 1999.

[36] M. Kafil and I. Ahmed, "Optimal Task Assignment in Heterogeneous Distributed Computing Systems," *Proc. IEEE Concurrency*, Vol. 6, No. 3, July-September 1998, pp. 42-51.

[37] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures," *Proc. ICPP,* IEEE-CS, v. 3, 1988.

[38] D.J. Kuck et. al., "Dependence Graphs and Compiler Optimizations," *Proc. 8th ACM Symposium on Principles of Programming Languages*, pp 207-218, Jan. 1981.

[39] Y. Kwok, I. Ahmad and J. Gu, "FAST: A Low-Complexity Algorithm for Efficient Scheduling of DAGs on Parallel Processors," *Proc. ICPP*, 1997.

[40] Y.K Kwok, "Parallel Program Execution on a Heterogeneous PC Cluster Using Task Duplication," *Proc. 9th HCW,* 364-374, 2000.

[41] D.Li and N.Ishii, "Scheduling task graphs onto heterogeneous multiprocessors," *Proc. IEEE Region 10's Ninth Annual International Conference. Theme: 'Frontiers of Computer Technology' (TENCON '94)*, pp. 556-563 vol.2, Aug. 1994.

[42] Y. A. Li and J. K. Antonio, "Estimating the execution time distribution for a task graph in a heterogeneous computing system," *Proc.6th Heterogeneous Computing Workshop (HCW '97)*, p.172, April 1997.

[43] Z.Liu, "Scheduling of random task graphs on parallel processors," *Proc. Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (MASCOTS '95), pp. 143-147, Jan. 1995.

[44] S.Y. Lee and J.Huang, "A Theoretical Approach to Load Balancing of a Target Task in a Temporally and Spatially Heterogeneous Grid Computing Environment," *Proc.GRID 2002*, pp. 70-81.

[45] Z. Liu, B. Fang, Y.Zhang and J.Tang "Scheduling algorithms for a fork DAG in a NOWs," *Proc. Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, Vol. 2* , pp. 959-960, May 2000.

[46] M. Maheswaran and H. J. Siegel, "A Dynamic Matching and Scheduling Algorithm for Heterogenous Computing Systems," *Proc.7th HCW*, pp. 57-69, IEEE Press, Mar. 1998.

[47] H. Oh and S. Ha, "A Static Scheduling Heuristic for Heterogeneous Processors," *Proc. Euro-Par*, pp. 573-577, v 2, 1996.

[48] S. S. Pande, D. P. Agrawal and J. Mauney, "A scalable scheduling method for functional parallelism on distributed memory multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, Vol. 6, No. 4, April 1995, pp. 388-399.

[49] C. Papadimitriou and M. Yannakakis, "Towards an Architecture Independent Analysis of Parallel Programs," *SIAM J. of Computing*, v 19, no. 2, pp 322-328, 1990.

[50] G.L Park, B.Shirazi, J.Marquis and H.Choo, "Decisive path scheduling: a new list scheduling method," *Proc. International Conference on Parallel Processing*, pp. 472-480, Aug. 1997

[51] A. Radulescu and A.J.C. Van Gemund, "Fast and Effective Task Scheduling in Heterogeneous Systems," *Proc. HCW, pp.229-238, May, 2000.*

[52] A. Ranaweera and D. P. Agrawal, "A Task Duplication Based algorithm for Heterogeneous Systems," *Proc. IPDPS*, pp 445-450, May 1-5, 2000.

[53] H. E.Rewini and T. G. Lewis, "Scheduling parallel programs onto arbitrary target architecture," *Journal of Parallel and Distributed Computing*, Vol. 9, No. 2, June 1990, pp. 138-153.

[54] P.C. SaiRanga and Sanjeev Baskiyar, "A Low-Complexity Algorithm for Dynamic Matching and Scheduling of Independent Tasks onto Heterogeneous Computing Systems," *Proc. ACMSE 2005*, March 2005.

[55] V.Sarkar, "Partitioning and Scheduling Parallel Programs for Multiprocessors," *The MIT Press*, Cambridge, MA 1989.

[56] G. Sih and E. Lee, "A Compile Time Scheduling Heuristic for Interconnection Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel and Distributed Systems,* vol. 4(2), pp. 175-187, 1993.

[57] H. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien, "The MicroGrid: A Scientific Tool for Modeling Computational Grids," *Proc. IEEE Supercomputing (SC 2000)*, Nov. 4-10, 2000, Dallas, USA.

[58] H. Topcuoglu, S. Hariri and M.-Y. Wu, "Task Scheduling Algorithms for Heterogeneous Processors," *Proc. HCW*, pp 3-14, 1999.

[59] H. Topcuoglu, S. Hariri, and M-Y. Wu "Performance-effective and low-complexity task scheduling for heterogeneous computing Parallel and Distributed Systems," *IEEE Trans. Parallel and Distributed Systems,* Volume: 13 Issue: 3, Mar 2002.

[60] T. Tsuchiya, T. Osada, T. Kikuno, "A new heuristic algorithm based on GA's for multiprocessor scheduling with task duplication," *Proc. Third International Conference on Algorithms and Architectures for Parallel Processing, 1997*, pp. 295- 308.

[61] J. Ullman, "NP-complete Scheduling Problems," *Proc. JCSS,* vol. 10, pp. 384-393. 1975.

[62] L.Yang, M. Jennifer and I.Foster, "Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments," *Proc. Supercomputing'03*, November 2003.

[63] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans. Parallel and Distributed Systems,* v. 5, no. 9,1994.