

**T.C.  
MARMARA ÜNİVERSİTESİ  
FEN BİLİMLERİ ENSTİTÜSÜ**

**RELIABILITY BASED DYNAMIC SCHEDULING OF  
INDEPENDENT TASKS IN HETEROGENEOUS  
COMPUTING ENVIRONMENTS**

**Esmâ YILDIRIM  
(Computer Engineer)**

**THESIS  
FOR THE DEGREE OF MASTER OF SCIENCE  
IN  
COMPUTER ENGINEERING PROGRAMME**

**SUPERVISOR  
Assoc.Prof.Dr. Haluk TOPÇUOĞLU**

**İSTANBUL 2006**

**T.C.  
MARMARA ÜNİVERSİTESİ  
FEN BİLİMLERİ ENSTİTÜSÜ**

**RELIABILITY BASED DYNAMIC SCHEDULING OF  
INDEPENDENT TASKS IN HETEROGENEOUS  
COMPUTING ENVIRONMENTS**

**Esmâ YILDIRIM  
(Computer Engineer)  
(141100320040255)**

**THESIS  
FOR THE DEGREE OF MASTER OF SCIENCE  
IN  
COMPUTER ENGINEERING PROGRAMME**

**SUPERVISOR  
Assoc.Prof.Dr. Haluk TOPÇUOĞLU**

**İSTANBUL 2006**

## **ACKNOWLEDGEMENTS**

I would like to thank my advisor Assoc. Prof. Haluk Topçuođlu for his outstanding guidance and support. It has been a great opportunity to work with him. I thankfully acknowledge Prof. Dr. M. Akif Eyller and Prof. Dr. Murat Dođruel for serving on my defense committee.

I would also like to thank my parents and sisters for their endless patience and pure love in every moment I needed.

**July 2006**

**Esma Yıldırım**

# TABLE OF CONTENTS

	<u>PAGE NO</u>
ACKNOWLEDGEMENTS .....	i
TABLE OF CONTENTS .....	ii
ÖZET .....	iv
ABSTRACT .....	v
LIST OF SYMBOLS/ABBREVIATIONS .....	vi
LIST OF FIGURES.....	viii
LIST OF TABLES.....	xii
<b>PART I.INTRODUCTION .....</b>	<b>1</b>
<b>I.1 OUTLINE OF THE THESIS .....</b>	<b>4</b>
<b>PART II.PROBLEM DEFINITION.....</b>	<b>5</b>
<b>II.1. SYSTEM MODEL .....</b>	<b>5</b>
<b>II.2. APPLICATION MODEL.....</b>	<b>6</b>
<b>II.3 RELIABILITY COMPUTATION .....</b>	<b>7</b>
<b>II.4. UNIFIED OBJECTIVE FOR MINIMIZING MAKESPAN AND     RELIABILITY COST IN DYNAMIC SCHEDULING .....</b>	<b>9</b>
<b>PART III.DYNAMIC SCHEDULING ALGORITMS.....</b>	<b>12</b>
<b>III.1. IMMEDIATE MODE ALGORITHMS .....</b>	<b>13</b>
III.1.1. MCT(Minimum Completion Time) Algorithm.....	13

III.1.2. MET(Minimum Execution Time) Algorithm .....	14
III.1.3. SA (Switching Algorithm) .....	14
III.1.4. KPB (K-Percent Best) Algorithm .....	16
<b>III.2. BATCH MODE ALGORITHMS .....</b>	<b>17</b>
III.2.1. Min-Min Algorithm .....	17
III.2.2. Max-Min Algorithm.....	18
III.2.3. Sufferage Algorithm.....	19
<b>PART IV.PROPOSED ALGORITMS .....</b>	<b>21</b>
<b>IV.1. RKPb (RELIABLE K-PERCENT BEST) ALGORITHM .....</b>	<b>21</b>
<b>IV.2. MEMETIC ALGORITHM WITH A UNIFIED OBJECTIVE .....</b>	<b>24</b>
IV.2.1. String Representation.....	24
IV.2.2. Initial Population Generation.....	25
IV.2.3. Ranking Based on Unified Objective metric .....	25
IV.2.4. Population Model.....	26
IV.2.5. Selection.....	27
IV.2.6. Local Search (Hill Climbing).....	27
IV.2.7. Crossover .....	28
IV.2.8. Mutation .....	30
IV.2.9. Outline of the algorithm.....	31
<b>PART V.EXPERIMENTAL STUDY .....</b>	<b>33</b>
<b>V.1. PERFORMANCE EVALUATION OF IMMEDIATE MODE</b>	
<b>ALGORITHMS.....</b>	<b>33</b>
V.1.1. Results and Discussion.....	34
<b>V.2. PERFORMANCE EVALUATION OF BATCH MODE ALGORITHMS</b>	<b>55</b>
V.2.1. RESULTS AND DISCUSSION .....	55
<b>PART VI.CONCLUSIONS.....</b>	<b>72</b>
<b>REFERENCES .....</b>	<b>73</b>

## ÖZET

### ÇOKTÜREL BİLGİSAYAR ORTAMLARINDA GÜVENİLİRLİK TABANLI DİNAMİK BAĞIMSIZ GÖREV ÇİZELGELEME

Çoktörel bir bilgisayar ortamı farklı işlem gücüne sahip birbirine bir ağ topolojisi ile bağlı kaynaklardan oluşur. Bu ortamda, dinamik olarak ulaşan bağımsız görevlerin çizelgelenmesi için çeşitli buluşsal algoritmalar geliştirilmiştir. Bu görevlerin ulaşım ve çalışma zamanları önceden bilinmemekte ve en kısa çizelgeleme uzunluğu hedeflenmektedir. Fakat ortamdaki kaynakların başarısız olma olasılığı ve bunun uygulama görevleri üzerindeki etkisi göz önüne alınmamaktadır. Bu olasılığı azaltmak için kaynakların güvenilirlik maaliyeti ve çizelgeleme uzunluğu aynı anda azaltılmalıdır. Bu çalışmada, dinamik çizelgeleme problemine hem çizelgeleme uzunluğu hem de güvenilirlik maaliyetini dikkate alan yeni birleşik bir formül geliştirilmiştir. Buna ek olarak iki çizelgeleme algoritması amaçlanmıştır. Yapılan test sonuçları göstermiştir ki iki hedefin önem ağırlığına göre çizelgeleme uzunluğu ve güvenilirlik maaliyetinin ikisi birden azaltılabilmektedir ve algoritmalarımız birçok durumda birleşik amaç doğrultusunda iyi sonuç vermiştir.

Anahtar kelimeler: Görev çizelgeleme, dinamik çizelgeleme, çoktörel hesaplama, güvenilirlik, evrimsel algoritmalar

Temmuz 2006

Esma Yıldırım

## **ABSTRACT**

### **RELIABILITY BASED DYNAMIC SCHEDULING OF INDEPENDENT TASKS IN HETEROGENEOUS COMPUTING ENVIRONMENTS**

A heterogeneous computing environment consists of resources with different processing powers connected via a network topology. There have been a number of heuristics proposed for the scheduling of dynamically arriving independent tasks in an heterogeneous computing environment. The arrival and execution times of those tasks are not known a priori and an objective of only minimization of makespan is target. However, the possibility of failure of the resources in the environment and its effects to the reliability of application tasks are not considered. To decrease the possibility of failures in the environment, both makespan and reliability cost of resources should be minimized. In this study, we propose a new unified objective of dynamic scheduling problem which considers both schedule length and reliability cost of resources. Additionally two scheduling algorithms (one for immediate mode and one for batch mode) are proposed as part of this study. The experimental results showed that according to the importance and weight of the two objectives in the algorithm, schedule length and reliability cost could be minimized together and our algorithms outperform the others dynamic scheduling algorithms in most of the cases in terms of unified objective.

Keywords: Task scheduling, dynamic scheduling, heterogeneous computing, reliability, evolutionary algorithms.

**July 2006**

**Esma Yıldırım**

## LIST OF SYMBOLS/ABBREVIATIONS

<b>HC</b>	: Heterogeneous Computing
<b>GA</b>	: Genetic Algorithm
<b>P</b>	: Set of processors
<b>L</b>	: Set of links
<b><math>l_i</math></b>	: $i$ th link
<b><math>p_i</math></b>	: $i$ th processor
<b><math>G(P,L)</math></b>	: Graph of processor set P connected by link set L
<b><math>P_{kl}</math></b>	: Path between $p_k$ and $p_l$
<b><math>\Lambda_i</math></b>	: Failure rate of $p_i$
<b><math>\Lambda_{kl}</math></b>	: Failure rate of link between $p_k$ and $p_l$
<b>T</b>	: Set of tasks
<b><math>t_i</math></b>	: $i$ th task
<b><math>te_{ij}</math></b>	: Expected(estimated) execution time of $t_i$ on $p_j$
<b><math>T_{max}</math></b>	: Task heterogeneity
<b><math>M_{max}</math></b>	: Machine heterogeneity
<b><math>te_i</math></b>	: Random number between 1 and $T_{max}$
<b><math>te_j</math></b>	: Random number between 1 and $M_{max}$
<b>m</b>	: Mean value for arrival time exponential distribution
<b><math>T_{met}</math></b>	: Mean execution time of tasks
<b><math>\alpha</math></b>	: Coefficient to change m
<b><math>R_j(T,X)</math></b>	: Reliability of $p_j$ for T under assignment X
<b><math>R_{kl}(T,X)</math></b>	: Reliability of $l_{kl}$ for T under assignment X
<b><math>R_k</math></b>	: Set of bridges and articulation points
<b><math>t_jA</math></b>	: Time at which last task on $p_j$ finishes execution
<b><math>t_{kl}A</math></b>	: Time at which last data items sended are finished on $l_{kl}$
<b><math>X_{ik}</math></b>	: Assignment of $t_i$ on $p_j$
<b><math>c_{ij}</math></b>	: Data to be transmitted between $p_i$ and $p_j$ in units
<b><math>w_{kb}</math></b>	: Transmissin rate of 1 kb
<b><math>S_k</math></b>	: Task set assigned to $p_k$
<b>p</b>	: Total number of processors
<b><math>ta_i</math></b>	: Arrival time of $t_i$
<b><math>I_{(i-1,i)}</math></b>	: Interarrival time between $t_{i-1}$ and $t_i$
<b><math>ts_{ij}</math></b>	: Start time of $t_i$ on $p_j$
<b><math>tr_j</math></b>	: Ready time of $p_j$
<b><math>tc_{ij}</math></b>	: Completion time of $t_i$ on $p_j$

$tc_i$	: Actual completion time of $t_i$
$M_1(X)$	: Minimization of makespan under assignment X
$M_2(X)$	: Minimization of reliability cost under assignment X
$w_1$	: Weight of makespan
$w_2$	: Weight of reliability cost
$tr_{max}$	: Maximum ready time in P
$tr_{min}$	: Minimum ready time in P
$\pi$	: Ratio of $tr_{max}$ and $tr_{min}$
$\pi_{low}$	: Low treshold for $\pi$
$\pi_{high}$	: High treshold for $\pi$
$k$	: Percentage of processors
$M_v$	: Set of tasks in a batch
$rank_u$	: Unified rank of processors
$rank_f$	: Rank of processors according to makespan
$rank_c$	: Rank of processors according to reliability cost
$S_{kpb}$	: k-percent best processors
$ch_i$	: Chromosome i
$avg_m$	: Average makespan of population
$avg_r$	: Average reliability cost of population

## LIST OF FIGURES

	<u>PAGE NO</u>
<b>Figure III.1:</b> Minimum Completion Time Algorithm.....	13
<b>Figure III.2:</b> Minimum Execution Time Algorithm.....	14
<b>Figure III.3:</b> Switching between MCT and MET.....	15
<b>Figure III.4:</b> Switching Algorithm.....	16
<b>Figure III.5:</b> K-Percent Best Algorithm.....	17
<b>Figure III.6:</b> Min-Min Algorithm.....	18
<b>Figure III.7:</b> Suffrage Algorithm.....	22
<b>Figure IV.1:</b> RKPB Algorithm.....	23
<b>Figure IV.2:</b> Sample string representation in partition approach.....	24
<b>Figure IV.3:</b> Chromosome structure.....	25
<b>Figure IV.4:</b> Tournament Selection.....	27
<b>Figure IV.5:</b> Application areas of local search in a hybrid GA.....	28
<b>Figure IV.6:</b> One point order crossover.....	29
<b>Figure IV.7:</b> Uniform Crossover.....	30
<b>Figure IV.8:</b> Mutation operators.....	31
<b>Figure IV.9:</b> Memetic Algorithm with a unified objective.....	32
<b>Figure V.1:</b> Makespan vs. Number of tasks, LoLo heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	35
<b>Figure V.2:</b> Reliability Cost vs. Number of tasks, LoLo heterogeneity, $w_1=0.25$ , $w_2=0.75$ ..	35
<b>Figure V.3:</b> Unified Objective vs. Number of tasks, LoLo heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	36
<b>Figure V.4:</b> Makespan vs. Number of tasks, LoLo heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	36
<b>Figure V.5:</b> Reliability Cost vs. Number of tasks, LoLo heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	36
<b>Figure V.6:</b> Unified Objective vs. Number of tasks, LoLo heterogeneity, $w_1=0.5$ , $w_2=0.5$ ..	37
<b>Figure V.7:</b> Makespan vs. Number of tasks, LoLo heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	37
<b>Figure V.8:</b> Reliability Cost vs. Number of tasks, LoLo heterogeneity, $w_1=0.75$ , $w_2=0.25$ ..	37
<b>Figure V.9:</b> Unified Objective vs. Number of tasks, LoLo heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	38
<b>Figure V.10:</b> Makespan vs. Number of tasks, HiHi heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	38
<b>Figure V.11:</b> Reliability Cost vs. Number of tasks, HiHi heterogeneity, $w_1=0.25$ , $w_2=0.75$ ..	38
<b>Figure V.12:</b> Unified Objective vs. Number of tasks, HiHi heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	39
<b>Figure V.13:</b> Makespan vs. Number of tasks, HiHi heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	39
<b>Figure V.14:</b> Reliability Cost vs. Number of tasks, HiHi heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	39

<b>Figure V.15:</b>	Unified Objective vs. Number of tasks, HiHi heterogeneity, $w_1=0.5$ , $w_2=0.5$ ...	40
<b>Figure V.16:</b>	Makespan vs. Number of tasks, HiHi heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	40
<b>Figure V.17:</b>	Reliability Cost vs. Number of tasks, HiHi heterogeneity, $w_1=0.75$ , $w_2=0.25$ ...	40
<b>Figure V.18:</b>	Makespan vs. Number of tasks, HiHi heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	41
<b>Figure V.19:</b>	Makespan vs. $\alpha$ , LoLo heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	42
<b>Figure V.20:</b>	Reliability Cost vs. $\alpha$ , LoLo heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	42
<b>Figure V.21:</b>	Unified Objective vs. $\alpha$ , LoLo heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	42
<b>Figure V.22:</b>	Makespan vs. $\alpha$ , LoLo heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	43
<b>Figure V.23:</b>	Reliability Cost vs. $\alpha$ , LoLo heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	43
<b>Figure V.24:</b>	Unified Objective vs. $\alpha$ , LoLo heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	43
<b>Figure V.25:</b>	Makespan vs. $\alpha$ , LoLo heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	44
<b>Figure V.26:</b>	Reliability Cost vs. $\alpha$ , LoLo heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	44
<b>Figure V.27:</b>	Unified Objective vs. $\alpha$ , LoLo heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	44
<b>Figure V.28:</b>	Makespan vs. $\alpha$ , HiHi heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	45
<b>Figure V.29:</b>	Reliability Cost vs. $\alpha$ , HiHi heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	45
<b>Figure V.30:</b>	Unified Objective vs. $\alpha$ , HiHi heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	45
<b>Figure V.31:</b>	Makespan vs. $\alpha$ , HiHi heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	46
<b>Figure V.32:</b>	Reliability Cost vs. $\alpha$ , HiHi heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	46
<b>Figure V.33:</b>	Unified Objective vs. $\alpha$ , HiHi heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	46
<b>Figure V.34:</b>	Makespan vs. $\alpha$ , HiHi heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	47
<b>Figure V.35:</b>	Reliability Cost vs. $\alpha$ , HiHi heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	47
<b>Figure V.36:</b>	Unified Objective vs. $\alpha$ , HiHi heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	47
<b>Figure V.37:</b>	Makespan vs. Number of processors, LoLo heterogeneity, $w_1=0.25$ , $w_2=0.75$ ..	48
<b>Figure V.38:</b>	Reliability Cost vs. Number of processors, LoLo heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	48
<b>Figure V.39:</b>	Unified Objective vs. Number of processors, LoLo heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	49
<b>Figure V.40:</b>	Makespan vs. Number of processors, LoLo heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	49
<b>Figure V.41:</b>	Reliability Cost vs. Number of processors, LoLo heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	49
<b>Figure V.42:</b>	Unified Objective vs. Number of processors, LoLo heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	50
<b>Figure V.43:</b>	Makespan vs. Number of processors, LoLo heterogeneity, $w_1=0.75$ , $w_2=0.25$ ..	50
<b>Figure V.44:</b>	Reliability Cost vs. Number of processors, LoLo heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	50
<b>Figure V.45:</b>	Unified Objective vs. Number of processors, LoLo heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	51
<b>Figure V.46:</b>	Makespan vs. Number of processors, HiHi heterogeneity, $w_1=0.25$ , $w_2=0.75$ ...	51
<b>Figure V.47:</b>	Reliability Cost vs. Number of processors, HiHi heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	51
<b>Figure V.48:</b>	Unified Objective vs. Number of processors, HiHi heterogeneity, $w_1=0.25$ , $w_2=0.75$ .....	52
<b>Figure V.49:</b>	Makespan vs. Number of processors, HiHi heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	52
<b>Figure V.50:</b>	Reliability Cost vs. Number of processors, HiHi heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	52

<b>Figure V.51:</b>	Unified Objective vs. Number of processors, HiHi heterogeneity, $w_1=0.5$ , $w_2=0.5$ .....	53
<b>Figure V.52:</b>	Makespan vs. Number of processors, HiHi heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	53
<b>Figure V.53:</b>	Reliability Cost vs. Number of processors, HiHi heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	53
<b>Figure V.54:</b>	Unified Objective vs. Number of processors, HiHi heterogeneity, $w_1=0.75$ , $w_2=0.25$ .....	54
<b>Figure V.55:</b>	Effect of weights on makespan.....	54
<b>Figure V.56:</b>	Effect of weights on Reliability Cost.....	55
<b>Figure V.57:</b>	Effect of weights on Unified Objective.....	55
<b>Figure V.2.1:</b>	Makespan vs. Number of tasks for batch size of 10, $W_1$ , LoLo.....	56
<b>Figure V.2.2:</b>	Reliability cost vs. Number of tasks for batch size of 10, $W_1$ , LoLo.....	57
<b>Figure V.2.3:</b>	Unified Objective vs. Number of tasks for batch size of 10, $W_1$ , LoLo.....	57
<b>Figure V.2.4:</b>	Makespan vs. Number of tasks for batch size of 10, $W_2$ , LoLo.....	57
<b>Figure V.2.5:</b>	Reliability cost vs. Number of tasks for batch size of 10, $W_2$ , LoLo.....	58
<b>Figure V.2.6:</b>	Unified Objective vs. Number of tasks for batch size of 10, $W_2$ , LoLo.....	58
<b>Figure V.2.7:</b>	Makespan vs. Number of tasks for batch size of 10, $W_3$ , LoLo.....	58
<b>Figure V.2.8:</b>	Reliability cost vs. Number of tasks for batch size of 10, $W_3$ , LoLo.....	59
<b>Figure V.2.9:</b>	Unified Objective vs. Number of tasks for batch size of 10, $W_3$ , LoLo.....	59
<b>Figure V.2.10:</b>	Makespan vs. Number of tasks for batch size of 10, $W_1$ , HiHi.....	59
<b>Figure V.2.11:</b>	Reliability cost vs. Number of tasks for batch size of 10, $W_1$ , HiHi.....	60
<b>Figure V.2.12:</b>	Unified Objective vs. Number of tasks for batch size of 10, $W_1$ , HiHi.....	60
<b>Figure V.2.13:</b>	Makespan vs. Number of tasks for batch size of 10, $W_2$ , HiHi.....	60
<b>Figure V.2.14:</b>	Reliability Cost vs. Number of tasks for batch size of 10, $W_2$ , HiHi.....	61
<b>Figure V.2.15:</b>	Unified Objective vs. Number of tasks for batch size of 10, $W_2$ , HiHi.....	61
<b>Figure V.2.16:</b>	Makespan vs. Number of tasks for batch size of 10, $W_3$ , HiHi.....	61
<b>Figure V.2.17:</b>	Reliability cost vs. Number of tasks for batch size of 10, $W_3$ , HiHi.....	62
<b>Figure V.2.18:</b>	Unified Objective vs. Number of tasks for batch size of 10, $W_3$ , HiHi.....	62
<b>Figure V.2.19:</b>	Makespan vs. Batch size for $W_3$ , HiHi.....	62
<b>Figure V.2.20:</b>	Reliability cost vs. Batch size for $W_3$ , HiHi.....	63
<b>Figure V.2.21:</b>	Unified Objective vs. Batch size for $W_3$ , HiHi.....	63
<b>Figure V.2.22:</b>	Makespan vs. Number of processors for batch size=25, $W_1$ , LoLo.....	64
<b>Figure V.2.23:</b>	Reliability Cost vs. Number of processors for batch size=25, $W_1$ , LoLo.....	64
<b>Figure V.2.24:</b>	Unified Objective vs. Number of processors for batch size=25, $W_1$ , LoLo.....	65
<b>Figure V.2.25:</b>	Makespan vs. Number of processors for batch size=25, $W_2$ , LoLo.....	65
<b>Figure V.2.26:</b>	Reliability cost vs. Number of processors for batch size=25, $W_2$ , LoLo.....	65
<b>Figure V.2.27:</b>	Unified Objective vs. Number of processors for batch size=25, $W_2$ , LoLo.....	66
<b>Figure V.2.28:</b>	Makespan vs. Number of processors for batch size=25, $W_3$ , LoLo.....	66
<b>Figure V.2.29:</b>	Reliability cost vs. Number of processors for batch size=25, $W_3$ , LoLo.....	66
<b>Figure V.2.30:</b>	Unified Objective vs. Number of processors for batch size=25, $W_3$ , LoLo.....	67
<b>Figure V.2.31:</b>	Makespan vs. Number of processors for batch size=25, $W_1$ , HiHi.....	67
<b>Figure V.2.32:</b>	Reliability cost vs. Number of processors for batch size=25, $W_1$ , HiHi.....	67
<b>Figure V.2.33:</b>	Unified Objective vs. Number of processors for batch size=25, $W_1$ , HiHi.....	68
<b>Figure V.2.34:</b>	Makespan vs. Number of processors for batch size=25, $W_2$ , HiHi.....	68
<b>Figure V.2.35:</b>	Reliability cost vs. Number of processors for batch size=25, $W_2$ , HiHi.....	68
<b>Figure V.2.36:</b>	Unified Objective vs. Number of processors for batch size=25, $W_2$ , HiHi.....	69

<b>Figure V.2.37:</b> Makespan vs. Number of processors for batch size=25, $W_3$ , HiHi.....	69
<b>Figure V.2.38:</b> Reliability cost vs. Number of processors for batch size=25, $W_3$ , HiHi.....	69
<b>Figure V.2.39:</b> Unified objective vs. Number of processors for batch size=25, $W_3$ , HiHi.....	70
<b>Figure V.2.40:</b> Makespan vs. Weights.....	70
<b>Figure V.2.41:</b> Reliability cost vs. Weights.....	71
<b>Figure V.2.42:</b> Unified Objective vs. Weights.....	71

## LIST OF TABLES

	<u>PAGE NO</u>
<b>Table IV.1:</b> Sample estimated execution time matrix.....	11
<b>Table IV.2:</b> Sample failure rates.....	12
<b>Table IV.3:</b> Ranks of processors.....	13

# PART I

## INTRODUCTION

A heterogeneous computing (HC) environment consists of machines with different processing powers and high-speed interconnection networks that can execute computationally intensive applications with diverse requirements. The problem of task scheduling is a key issue that has to be solved to utilize the resources of an HC environment. As the solutions to the problem are quite diverse, the problem itself could be categorized also. In some cases, the problem is solved statically where the knowledge about tasks and network topology is known before scheduling, which is called *static scheduling*. On the other hand, there is no priori information in *dynamic scheduling* where scheduling decisions are made during application execution. The application task graphs are also divided into two categories: dependent and independent. In dependent task graphs, tasks need data to be sent from other tasks hence their start time depends on the finish time of previously executed tasks. However, there is no connection between tasks in independent task graphs; thus more freedom is gained while applying a scheduling strategy. Many studies have been made on both dynamic[3,4,6,7] and static[1,2,5,8] scheduling category.

Dynamic independent scheduling problem in heterogeneous computing environments in which tasks arrive randomly and independently, is a well-known problem, studied extensively and is NP-complete. Much of studies that have been done

is based on some heuristic methods in [3,4,6,7] implemented considering totally dynamic environments where tasks execution times and network topology is not known a priori. Maheswaran et. al [3] compares many dynamic heuristics (which are immediate and batch modes) for independent tasks; and their objective is the minimization of the makespan as a sign of maximization of throughput. However, some conditions of tasks are to be taken into account such as task priorities and deadlines [4]. In both studies no objective for reliability maximization is targeted. Additionally, many other novel heuristics refer to the model given in [3] with some additional task selection mechanism. One example is to minimize makespan by using a batch mode heuristic and to select tasks according to a calculated weighted mean execution time [5].

On the other hand, dynamic scheduling heuristics are also proposed for tasks that have DAG structure and dependencies among themselves. Some of them are hybrid heuristics [6,8] which are a combination of static and dynamic scheduling strategies. In [6], tasks are divided into blocks according to dependencies among them such that all tasks in a block are independent from each other and a static scheduling heuristic is applied. The dynamism is added when the values of actual execution times arrives and a remapping occurs according to that new information. While in [6] blocks are divided according to the shape of the task graph, in [8] a new rank calculation is done which is similar in the rank calculation of HEFT algorithm [2] and blocks are divided according to that information. Then a dynamic independent scheduling algorithm is applied to the tasks in blocks. However as a new method this algorithm tries to decrease makespan with an additional refinement step. This thesis emphasizes on scheduling of independent tasks; therefore, DAG scheduling part is out of scope of this thesis.

Although there are GA-based solutions that provide good results, they consider static task scheduling model [31,32,33,34]. A new study [22] applies GA to a dynamic environment where tasks arrive dynamically and randomly in time. The algorithm works in batch fashion and targets to minimize only makespan. As a difference dynamic fitness calculation, changing batch size and rebalancing methods are used.

Although efficient scheduling decisions provide shorter running time of applications, machine and network failures are unavoidable in any heterogeneous

computing environment and a scheduling algorithm that minimizes only the schedule length may lead to a high failure probability [27]. There are several studies that describe reliability model of a system and reliability computation of tasks scheduling on different systems but almost all of them considers static environment and scheduling strategies[9,10,11,12,13]; and there are no research work which considers dynamic environments . A reliability computation model is given in [11,12,13] which is suitable for independent tasks but for only tree based networks. The reliability of machines and links are both taken into account.

Studies that consider to minimize multi objectives (i.e. minimization of both schedule length and failure probability) are so few and they are proposed for only static scheduling[24,25]. In [24], both makespan and reliability cost is tried to be minimized for DAG applications considering an arbitrarily connected network topology. The task execution times and network structure is known a priori. A new reliability model is proposed for arbitrarily connected networks and dependent applications where reliability of links is also a matter. Two new algorithms are presented in their paper[24]: an extension of the DLS algorithm and a GA based solution. Another multiobjective study is [26]; where they target to minimize processor number and makespan at the same time instead of reliability and makespan. The way two objectives are unified into one is a good way which can be applied considering the reliability factor.

In this study, we target to develop reliability-based dynamic scheduling algorithms running onto heterogeneous computing environment. In our application model we consider independent tasks. Both immediate mode and batch mode of dynamic scheduling are considered as part of this thesis. We propose two algorithms; RKPB(reliable k-percent best) and a MA(Memetic Algorithm). The first one is an immediate mode algorithm while the latter one is a batch mode algorithm. The testing results showed that our algorithms outperformed the other algorithms in terms of unified objective and reliability cost in most of the cases. The two objective can be minimized at the same time but with a trade-off between them.

## **I.1 OUTLINE OF THE THESIS**

The rest of this thesis is structured as follows. Chapter 2 describes the unified objective of dynamic scheduling problem which considers the makespan and the reliability cost. It also presents the details of reliability-based system and application models. We also formulate the equations to combine two given objectives at the end of the chapter. Chapter 3 summarizes algorithms in the literature which solves dynamic tasks scheduling problem for both immediate and batch modes.

Chapter 4 presents the details of our three algorithms for reliability-based dynamic scheduling. Two of these algorithms are heuristics that work in both immediate and batch modes. The last one is a memetic algorithm that works in batch mode.

We present the details of computational experiments for measuring the performance of our algorithms in Chapter 5. Finally Chapter 6 gives a summary of what has been discussed and proposed in our study.

## **PART II**

### **PROBLEM DEFINITION**

The problem considered in this study is developing a scheduling strategy in which both makespan and reliability cost components are taken into account and minimized considering a dynamic environment where network topology can change and tasks arrive randomly. In this chapter, system and application models are presented. Additionally, the reliability computation model is explained and a formulation is given to minimize both objectives.

#### **II.1. SYSTEM MODEL**

The network topology of the heterogeneous computing environment presented is an undirected graph  $G(P,L)$ , where  $P$  denotes a set of processing machines  $\{p_1, p_2, p_3, \dots, p_n\}$  and  $L$  denotes the set of links between those machines  $(l_1, l_2, \dots, l_m)$ . The processing machines are arbitrarily connected through links and the graph  $G$  can change dynamically in time, where some machines can turn off due to failures or user requests. Both machines and links are heterogeneous meaning that they have different processing power and links have different transmission rates. Each task has different execution times on different processors. It is also assumed that the links are contention free. Here  $p_{kl}$  represents a path from a source machine  $p_k$  to a destination machine  $p_l$ .

This path consists of a set of resources that could be machines or links which are not visited more than once.

Every machine  $p_i$  has a failure rate that follows a poisson distribution with a predefined mean and it is assumed to be independent. The failure rate of  $p_i$  is represented by  $\lambda_i$ . This distribution model does not always overlap to the actual failure rates of machines and links but still lead to advantageous mathematical models[24]. Also once a resource fails it stays in this mode thereafter and does not go on execution of other tasks.

## II.2. APPLICATION MODEL

An application consists of tasks that are independent from each other and having no data dependency among themselves. Tasks arrive randomly to the system and their execution times are not known a priori thus presenting a dynamic characteristics. Let  $T$  be a set of independent tasks  $\{t_1, t_2, t_3, \dots, t_n\}$ . The execution times of tasks arriving to the system is assumed to follow a uniform distribution. The expected execution time of a task is the amount of time of taken by  $p_j$  to execute  $t_i$  when  $p_j$  has no load. This time is different based on the machine it is executed and it is set based on task heterogeneity factor and machine heterogeneity factor. There are four different alternatives in our experiment related with heterogeneity factors which are HiHi(high task high machine heterogeneity), HiLo(high task low machine heterogeneity), LoHi and LoLo. In the following equation

$$te_{ij} = te_i \times te_j \quad (II.1)$$

$te_i$  is a uniformly distributed random number between 1 and maximum task heterogeneity value  $T_{max}$  and  $te_j$  is again a uniformly distributed number between 1 and maximum machine heterogeneity value  $M_{max}$ . The greater  $T_{max}$  and  $M_{max}$  the greater the task and machine heterogeneity.

A matrix is constructed to show each task's  $t_{eij}$  where rows show the tasks and columns show the computational machines. The matrix is set to be consistent if a machine executes a task faster than other machines then it executes all the tasks faster than other machines. On the other hand a matrix is set to be inconsistent if the above rule does not hold. To make the matrix consistent the values generated are sorted.

The interarrival time between tasks follows an exponential distribution with some mean  $m$ . But this  $m$  value should depend on the characteristics of tasks execution time on machines. The following formulation is used to calculate  $m$ .

$$m = (T_{met} / p) \times \alpha \quad (II.2)$$

where  $\alpha$  is a coefficient to increase and decrease task arrival rate and  $T_{met}$  is mean execution time of tasks which is calculated as follows:

$$T_{met} = (T_{max} \times M_{max} + 1/2) \quad (II.3)$$

The greater the value of  $\alpha$ , the later tasks arrive to the system.

The assumption is kept that a task will run to completion without preemption when it is assigned to a computational machine. The tasks are scheduled to processors by a centralized scheduler with the goal of minimizing two objectives.

## II.3 RELIABILITY COMPUTATION

The reliability of a machine is defined in the literature as the probability that the machine is functional for the time interval in which tasks are executing on it. It is denoted as  $R_j(T,X)$  where  $j$  is the processor number,  $T$  is the task set assigned and  $X$  is the task-processor assignment. The same definition is also applicable for the reliability of a link and denoted as  $R_{k,l}(T,X)$  which means the reliability of the link that connects processor  $k$  and  $l$ . There are different reliability models presented for different scheduling problems. In a previous work[24] the reliability model is presented for arbitrarily connected networks and DAG structured task graphs where some tasks depend on the data other tasks will send after finishing executing. The route of the data

to be sent is important in the sense that different routes could have different reliability costs. But it is difficult to calculate which path and processor will give the least reliability cost. Hence only bridges and articulation points as well as source and destination processors are taken into account. This model is given in the following formulation. Because the reliability of a machine or link  $i$  is assumed to follow a Poisson process it is equal to  $e^{-\lambda_i t}$  at time  $t$ .

$$R(T, X) = \prod_{p_j \in R_k} R_j(T, X) \cdot \prod_{l_{k,l} \in R_k} R_{k,l}(T, X) \quad (\text{II.4})$$

$$R(T, X) = \prod_{p_j \in R_k} e^{-\lambda_j t_j^A} \cdot \prod_{l_{k,l} \in R_k} e^{-\lambda_{k,l} t_{k,l}^A}$$

$$R(T, X) = e^{-COST(X)}$$

$$COST(X) = \sum_{p_j \in R_k} \lambda_j t_j^A \cdot \sum_{l_{k,l} \in R_k} \lambda_{k,l} t_{k,l}^A \quad (\text{II.5})$$

Here  $R_k$  is the set of bridges and articulation points that connect the topology and are assigned tasks or data to be transmitted as well as the source and destination machines. This kind of calculation is not necessary because we think our environment as dynamic and tasks are independent from each other thus no data communication exists between the processors executing two different tasks. In this case the topology of the network also loses its importance.

The time interval that a machine should be functional is assumed to be the time the machine starts at time 0 and finishes the last task assigned to it and denoted as  $t_j^A$ . As machines can turn on and off in our dynamic environment there is also no need that the machine should be functional until it finishes all tasks assigned to it. The idle times of machines should not be taken into account. It only needs to be functional from the time it starts executing a task to the time it finishes the execution of the same task.

In Shatz et al.'s[11] model, the cost function is defined as

$$COST(X) = \sum_{k=1}^p \sum_{i=1}^T \lambda_k x_{ik} t e_{ik} + \sum_{k=1}^{p-1} \sum_{b>k} \sum_{i=1}^T \sum_{j=1}^T \lambda_{kb} x_{ik} (c_{ij} / w_{kb}) \quad (\text{II.6})$$

where  $p$  is processor number,  $T$  is task number,  $\lambda_k$  is the failure probability of machine  $k$ ,  $x_{ik}$  is assignment of task  $i$  to machine  $k$  (which only 0 or 1 value),  $t_{ik}$  is the execution of task  $i$  on machine  $k$ ,  $\lambda_{kb}$  is the failure probability of link  $l$  kb,  $c_{ij}$  is the data to be transmitted in units and  $w_{kb}$  is the transmission rate of  $l$  kb. Because tasks assigned are part of a module, the intermodule communication occurs between tasks thus making the reliability of links important which is not in our case.

The reliability model used in our own study does not include the reliability of links but only processors which execute tasks independently. Hence the model is reduced to the following formulation.

$$R(T, X) = \prod_{k=1}^p R_k(T, X) = e^{(-COST(X))} \quad (II.7)$$

Where  $p$  is the total number of processors,  $T$  is the task set under assignment  $X$  which is equal to  $e^{(-COST(X))}$ .

$$COST(X) = \sum_{k=1}^p \sum_{S_k(i)} \lambda_k t_{ik} \quad (II.8)$$

where  $p$  is the total processor number,  $\lambda$  is failure probability of  $p_k$ ,  $S_k$  is task set assigned to  $p_k$ ,  $t_{ik}$  is execution time of task  $i$  on processor  $k$ .

## II.4. UNIFIED OBJECTIVE FOR MINIMIZING MAKESPAN AND RELIABILITY COST IN DYNAMIC SCHEDULING

Foregoing studies have shown that an objective for minimization of makespan only results in high reliability costs while an objective for minimization of reliability cost results in very high makespans. Thus in this study both makespan and reliability costs are tried to be minimized for dynamic task scheduling.

Before the explanation of formulation that minimizes both objectives some terms related to that formulation must be explained. As it is said before the expected execution time of a task on a machine is denoted by  $te_{ij}$ . The arrival time of a task is the time the task reaches to the environment and placed on scheduler machine's queue. It is denoted as  $ta_i$  and calculated as the summation of the arrival time of the previous task and interarrival time  $I_{(i-1, i)}$ .

$$ta_i = ta_{(i-1)} + I_{(i-1, i)} \quad (II.9)$$

The ready time of a machine is denoted by  $tr_j$  and is the finish time of the last task assigned to that machine. The start time of a task on a machine is maximum value of machine ready time and arrival time of that task. It is denoted as  $ts_{ij}$ .

$$ts_{ij} = \max(ta_i, tr_j) \quad (II.10)$$

The completion time of a task on a machine is denoted by  $tc_{ij}$  and is the summation of start time and expected execution time of that task on the considered machine.

$$tc_{ij} = ts_{ij} + te_{ij} \quad (II.11)$$

The calculated makespan is the actual completion time of a task that finishes latest among a predefined number of tasks to be executed. It is denoted as

$$Makespan = \max_{t_i \in T} (tc_i) \quad (II.12)$$

where  $tc_i$  is the actual completion time of a task and  $T$  is the set of tasks to be executed.

Our first objective is to first minimize the makespan under task assignment  $X$  and denoted as  $M_1(X)$  and second objective is to minimize reliability cost under task assignment  $X$  and denoted by  $M_2(X)$ . Both objective can not be achieved as their optimal value but some weight ratio could be assigned to them to construct a unified objective  $M$ .

$$M(X) = w_1 M_1(X) + w_2 M_2(X) \quad (\text{II.13})$$

$w_1$  and  $w_2$  are weights assigned to the objectives. They take values in the range  $[0,1]$ . As  $w_1$  increases and  $w_2$  decreases, the makespan of total number of tasks shortens while the reliability cost is increased. The inverse occurs when  $w_2$  increases with respect to  $w_1$ .

## **PART III**

### **DYNAMIC SCHEDULING ALGORITHMS**

The dynamic scheduling algorithms presented in the literature work in two modes: Immediate and batch[3]. An immediate mode algorithm schedules only the newly arrived tasks. A task is scheduled as soon as it arrives to the scheduler's queue. The characteristics of the previously arrived tasks or yet to be arrived tasks are not important. The knowledge of the current task is used to schedule it. This knowledge consists of tasks arrival time, processor ready time and tasks execution times on active processors.

However in batch mode, tasks are not scheduled as soon as they arrive but collected in a batch. In this case the knowledge about more than one task is known and a better scheduling could be made based on that knowledge. The total of tasks in a batch is behaved as a meta task and a scheduling event is raised for that metatask.

Another important issue about batch mode heuristics is the batch size. In order to decide batch size there are two methods to be used: a) regular time interval, b) fixed count. In the former case, the metatask consists of tasks that arrives and collected in the batch in a predefined time interval. When this period is over a mapping event is raised. The number of tasks is depends on the arrival rate of tasks. In the later case, a fixed number of tasks exist in a batch and the mapping event is raised as soon as that number of tasks arrive. This time again depends on the arrival rate of tasks. For example it may

take 2 seconds for 10 tasks to be collected in a batch at one time while at some other time it could take 20 seconds.

### III.1. IMMEDIATE MODE ALGORITHMS

This section gives details of four immediate mode algorithms that are presented in the literature which are for dynamic scheduling of independent tasks.

#### III.1.1. MCT(Minimum Completion Time) Algorithm

The MCT algorithm[3] maps a newly arrived task to the machine that completes the task in minimum time. The calculation of that time is based on the arrival time, processor ready time and execution time on that machine. However, in this mapping there is a probability that the task is not assigned to its best execution time machine but rather a slow one. The algorithm takes  $O(p)$  time for only one task where  $p$  is total number of processors.

```
(1)   for each newly arriving task  $t_i$ 
(2)        $min = \infty$ 
(3)        $m = -1$ 
(4)       for each processor  $p_j$ 
(5)           Calculate  $tc_{ij}$  based on previous formulation
(6)           if  $tc_{ij}$  is less than  $min$ 
(7)                $min = tc_{ij}$ 
(8)                $m = j$ 
(9)       endfor
(10)      Assign task  $i$  to processor  $m$ 
(11)      Update the ready time of  $p_m$ 
(12)      endfor
```

Figure III.1: Minimum Completion Time Algorithm[3]

### III.1.2. MET(Minimum Execution Time) Algorithm

The MET algorithm[3] assigns a newly arriving task to the machine that executes it in the least amount of time. The machine ready times are not taken into consideration. The disadvantage of this heuristic is that some machines would be heavily loaded. In the case of inconsistent execution time matrices this unbalance is not so severe. But for consistent and semi consistent matrices, some processors will always execute tasks faster than other machine thus assigned every task. In some cases this could even lead the system to behave as a uniprocessor system. As in the previous case this algorithm also takes  $O(p)$  time for only one task.

```
(1)  for each newly arriving task  $t_i$ 
(2)       $min = \infty$ 
(3)       $m = -1$ 
(4)      for each processor  $p_j$ 
(5)          if  $te_{ij}$  is less than  $min$ 
(6)               $min = te_{ij}$ 
(7)               $m = j$ 
(8)          endfor
(9)      Assign task  $i$  to processor  $m$ 
(10)     Update the ready time of  $p_m$ 
(11) endfor
```

Figure III.2: Minimum Execution Time Algorithm[3]

### III.1.3. SA (Switching Algorithm)

The SA algorithm[3] both uses MCT and MET at the same time but according to some condition met. For example, among 100 tasks to be assigned, SA could assign 40 of them according to MCT, then switches and assigns 20 of them with MET and finally returns to MCT again to assign the rest 40 tasks.

The reason to use both algorithms at the same time is to benefit from their good qualities. While MET choses best execution time machines and unbalance the load,

MCT tries to balance it. To decide on when a switch will occur the load of the system must be calculated at each cycle. In order to do this we need to find maximum and minimum ready times among processors and assign it to  $tr_{max}$  and  $tr_{min}$ . Then a ratio between them is used to determine the load which is denoted by  $\pi$  and called the load balance index[3]. It is the ratio of  $tr_{min} / tr_{max}$ . The value of  $\pi$  ranges between 0 and 1. If this value is too small, that means there is a severe unbalance in the system and if it is close to 1 then there is a load balance. There are two threshold values to determine the balance : $\pi_{high}$  and  $\pi_{low}$ .

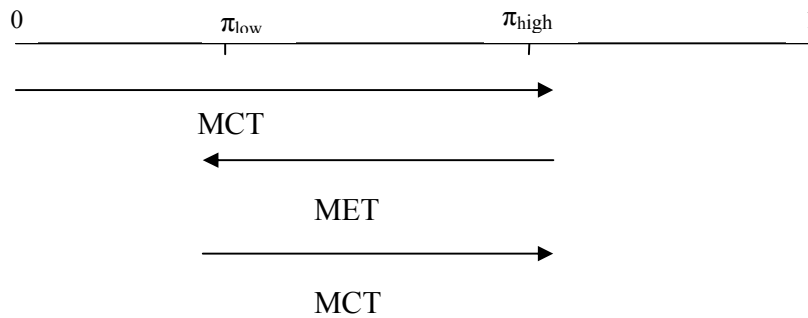


Figure III.3: Switching between MCT and MET

The heuristics starts with MCT and continues until  $\pi$  reaches  $\pi_{high}$ . In this case the load on the system is quite balanced. Then it switches to MET and  $\pi$  value starts to decrease towards  $\pi_{low}$  and the load is unbalanced. When it passes this threshold it then returns to MCT and the cycle goes on like this.

```

(1)  for each newly arriving task  $t_i$ 
(2)       $tr_{max}$ =maximum ready time among processors
(3)       $tr_{min}$ =minimum ready time among processors
(4)       $\pi = tr_{min}/tr_{max}$ 
(5)      switchmode = 0
(6)      if  $\pi$  is greater than  $\pi_{high}$ 
(7)          switchmode = 1
(8)      if  $\pi$  is less than  $\pi_{low}$ 
(9)          switchmode = 0
(10)     if switchmode = 1
(11)         apply MET (rows 2-10)
(12)     else
(13)         apply MCT (rows 2-11)
(14) endfor

```

Figure III.4: Switching Algorithm[3]

### III.1.4. KPB (K-Percent Best) Algorithm

The KPB algorithm[3] maps tasks to only a selected portion processor set P. The selected processors are the ones that execute the task in shortest time interval. The percentage of those processors is denoted by k. Among those processors the one that gives the minimum completion time is selected for assignment.

While SA tries to apply good features of MET and MCT separately, KPB does it at the same time and simultaneously. The reason for that is by selecting the best k-percent processors it prevents the assignment of task to a poor machine and by applying MCT it keeps the load in balance. For values of  $k=100$  the algorithm turns into MCT and  $k=100/p$  it turns into MET.

```

(1)  for each newly arriving task  $t_i$ 
(2)      Sort processors in  $P$  in increasing order of  $te_{ij}$  values
(3)       $S_{kpb}$ =The first  $k*p/100$  processors in  $P$ 
(4)       $min = \infty$ 
(5)       $m = -1$ 
(6)      for each processor  $p_j$  in  $S_{kpb}$ 
(7)          if  $tc_{ij}$  less than  $min$ 
(8)               $min=tc_{ij}$ 
(9)               $m=j$ 
(10)     endfor
(11)     Assign task  $i$  to processor  $m$ 
(12)     Update the ready time of  $p_m$ 
(13)     endfor

```

Figure III.5: K-Percent Best Algorithm[3]

The sorting operation in the second step of the algorithm takes at least  $O(p \log_p)$ . The rows 6-10 takes  $O(p)$ . In total the complexity of the algorithm is  $O(p \log_p)$ .

## III.2. BATCH MODE ALGORITHMS

This section gives details of the batch mode algorithms for dynamic scheduling of independent tasks.

### III.2.1. Min-Min Algorithm

This algorithm[3] calculates the minimum completion times of all tasks in the batch which is called first Min. Then among these tasks it chooses the task that gives the minimum earliest completion time. That operation refers to the second Min in the name of the algorithm. The chosen task is assigned to its minimum completion time processor. The ready time of the processor and task completion times for that machine are updated. The procedure continues until all the tasks in the batch are assigned. Because processor ready times change less, Min-Min algorithm is expected to give

good results. The running time of this algorithm is  $O(V^2p)$  where  $V$  is the number of tasks in batch and  $p$  is the total processor number.

```

(1)   for each task  $t_i$  in meta-task  $M_v$ 
(2)       for each processor  $j$ 
(3)            $tc_{ij} = te_{ij} + tr_j$ 
(4)   do until all tasks in  $M_v$  are mapped
(5)       for each task  $t_i$  in  $M_v$ 
(6)            $tc_i = \infty$ 
(7)           assigned_proc = -1
(8)       for each processor  $j$ 
(9)           if  $tc_{ij}$  is less than  $tc_i$ 
(10)               $tc_i = tc_{ij}$ 
(11)              assigned_proc =  $j$ 
(12)           end for
(13)       end for
(14)       for each task  $t_i$  in  $M_v$ 
(15)           find the minimum  $tc_i$ 
(16)           Assign task  $m$  to its assigned_proc $m$ 
(17)           Delete task  $m$  from  $M_v$ 
(18)           Update ready time of assigned_proc $m$  and task completion times for
              that processor.
(19)   End do

```

Figure III.6: Min-Min Algorithm[3]

### III.2.2. Max-Min Algorithm

The Max-Min [3] algorithm is similar to Min-Min. The only difference is among the tasks in the batch the one that give the maximum earliest completion time is selected first and assigned to its earliest completion time processor.

Max-Min algorithm have advantages over Min-min in some cases where the number of long tasks are so less than the number of short tasks. Because Max-Min

schedules the long tasks first and while it is executed it can schedule many short tasks. However Min-Min will schedule short tasks first and then the long tasks will be executed thus gives longer makespan.

### **III.2.3. Sufferage Algorithm**

The difference between the completion times of a task when it is assigned to its first best processor that gives minimum completion time and its second best processor is called the Sufferage value of that task[3]. If the best processor of more than one task in the batch is the same, the one of which sufferage value is greatest is assigned to its best processor.

This algorithm first calculates the completion times of tasks on each processor. Then a loop starts until all task in the batch are assigned. All the processors are marked as unassigned. Then for each task, the first and second minimum completion times and the processor that gives this result are found. Their sufferage value is calculated. If the best processor of the selected task is unassigned then the task is assigned to its best processor. If not the previously assigned task's sufferage value is compared to the selected task. If the sufferage value of the current task is greater than the previously assigned task to that processor, that task is marked as unassigned and added to the batch again. So the current task is assigned to its best processor. The task is deleted from the batch. The processor ready times and completion times of tasks for processors are updated and the loop goes on for other unassigned tasks in the batch.

```

(1) for each task  $t_i$  in meta-task  $M_v$ 
(2)   for each processor  $j$ 
(3)      $tc_{ij} = te_{ij} + tr_j$ 
(4)   do until all tasks in  $M_v$  are mapped
(5)     mark all processors as unassigned
(6)     for each task  $t_i$  in  $M_v$ 
(7)       find the processor  $p_j$  that gives minimum completion time
(8)       sufferage value = second minimum completion time - first minimum
completion time
(9)       if  $p_j$  is unassigned
(10)        assign  $t_i$  to  $p_j$ , delete  $t_i$  from  $M_v$  and mark  $p_j$  as assigned
(11)       else
(12)        if sufferage value of task  $t_i$  is greater than the sufferage value of task  $t_l$  that
is already assigned to  $p_j$ 
(13)         unassign  $t_l$ , add  $t_l$  to  $M_v$  ,
(14)         assign  $t_i$  to  $p_j$ 
(15)         delete  $t_l$  from  $M_v$ 
(16)       end for
(17)     update processor ready times
(18)   update  $tc$  matrix
(19) end do

```

Figure III.8: Sufferage Algorithm[3]

## **PART IV**

### **PROPOSED ALGORITHMS**

In this section we propose two different algorithms for our unified objective of minimizing both makespan and reliability cost. One of them is for immediate mode scheduling and the other is for batch mode scheduling. The details of the algorithms will be given in the following sections.

#### **IV.1. RKP (RELIABLE K-PERCENT BEST) ALGORITHM**

Among the immediate mode algorithms KPB is the one that gives better results in terms of makespan but does not take reliability cost into consideration. In our algorithm we are able to minimize both of them simultaneously. To make sure that makespan does not increase too much, we keep to assign tasks only to their k-percent best processors in terms of estimated execution time. Among those processors a decision must be made according to a given rank. This rank is calculated as follows.

The completion time of the task on every processor in the set  $S_{kpb}$  are calculated which only contains k-percent best processors for that task and the processors are sorted in ascending order in terms of completion time values. The processor that gives the minimum completion time value is assigned rank 1 and ranks are incremented by 1 for

the other processors in the set. The maximum rank value is the number of processors in the set. This rank is in terms of completion time and denoted by  $rank_c$ .

The processors must also be given another rank for the reliability cost. The reliability cost of assigning one task to a processor is multiplication of the estimated execution time of that task on the proposed processor and the failure rate of that processor. Then the processors are sorted in ascending order of the calculated reliability cost values and given ranks. That rank is denoted by  $rank_f$ . Again the ranks varies between 1 and  $k \cdot p / 100$ . The processor with least reliability cost is assigned the rank 1.

So far each processor is given two different ranks for the current task. To unify our two objectives into one, we must give processors a unified rank. It is impossible to find optimum values for each objective at the same time but their weights in the solution could be shared. Let  $w_1$  and  $w_2$  be two constants between values 0 and 1.  $w_1$  denotes the weight of the first objective  $M_1(X)$  and  $w_2$  denotes the weight of the second objective  $M_2(X)$ . The unified rank is denoted by  $rank_u$  equal to:

$$rank_u = rank_c \times w_1 + rank_f \times w_2 \quad (IV.1)$$

The task is then assigned to the processor with the minimum unified rank value. An example execution time matrix is given in Table III.1 and failure probabilities of processors are given in Table III.2.

<b>Task/Processor</b>	<b>p<sub>1</sub></b>	<b>P<sub>2</sub></b>	<b>p<sub>3</sub></b>	<b>P<sub>4</sub></b>	<b>p<sub>5</sub></b>
<b>t<sub>1</sub></b>	11.2	57.5	20.4	54.2	71.3
<b>t<sub>2</sub></b>	28.5	15.7	41.4	72.4	19.2

Table IV.1: Sample estimated execution time matrix

	<b>p<sub>1</sub></b>	<b>p<sub>2</sub></b>	<b>p<sub>3</sub></b>	<b>p<sub>4</sub></b>	<b>p<sub>5</sub></b>
<b>Λ</b>	2.0	1.5	0.5	1.0	4.0

Table IV.2: Sample failure rates

At time<sub>0</sub>,  $t_1$  arrives to the system and processors are sorted as follows according to minimum completion time and reliability cost values. For  $k=60$  which is the

percentage value of processor set , only three processors are selected which gives the fastest execution times for task 1. Those are  $p_1$ ,  $p_3$  and  $p_4$ .

	$p_1$	$p_3$	$p_4$
$rank_c$	1	2	3
$rank_f$	2	1	3
$rank_u$	1.75	1.25	3

Table IV.3: Ranks of processors

For  $w_1=0.25$  and  $w_2=0.75$   $p_3$  is the one that has the lowest rank value. Thus task<sub>1</sub> is assigned to  $p_3$ .

- (1) **for** each newly arriving task  $t_i$
- (2)     Sort processors in  $P$  in increasing order of  $te_{ij}$  values
- (3)      $S_{kpb}$ =The first  $k*m/100$  processors in  $P$
- (4)     Sort processors in  $S_{kpb}$  according to minimum completion time values for  $t_i$
- (5)      $r=1$
- (6)     For each  $p_i$  in  $S_{kpb}$
- (7)          $rank_{c_i}=r$
- (8)          $r=r+1$
- (9)     Sort processors in  $S_{kpb}$  according to reliability cost values for  $t_i$
- (10)      $r=1$
- (11)     **for** each  $p_i$  in  $S_{kpb}$
- (12)          $rank_{f_i}=r$
- (13)          $r=r+1$
- (14)     **for** each  $p_i$  in  $S_{kpb}$
- (15)          $rank_{u_i}=rank_{c_i} * w_1 + rank_{f_i} * w_2$
- (16)     Find the processor  $m$  which gives the minimum  $rank_{u_i}$  value
- (17)     Assign  $t_i$  to  $p_m$
- (18)     Update ready time of  $p_m$
- (19) **end for**

Figure IV.1: RKP B Algorithm



In our algorithm we use the assignment approach. Each assignment of task to processor is represented by *genes*. The string of genes collected together represents the *chromosome* structure. The order of genes in the chromosome represents the execution order of tasks on assigned processors. The size of the chromosome is the number of tasks in the batch. According to Figure IV.3, task 1 is assigned to processor 4, task 5 to processor 3, task 1 to processor 3 and task 4 to processor 2.

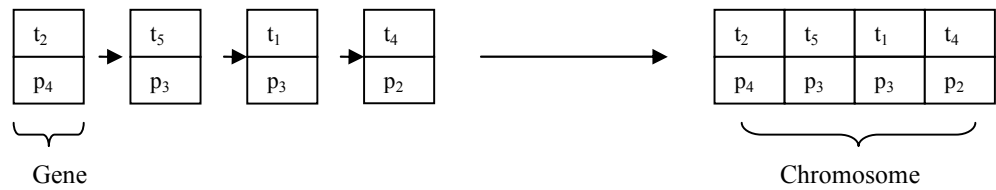


Figure IV.3: Chromosome structure

## IV.2.2. Initial Population Generation

In our algorithm we combine seeding and local search methods to generate the initial population. Seeding is the addition of one or more good solutions to the population and local search is applied on each individual to include some optimal set of points in the population. In seeding phase, one of the individuals is created as a result of MCT algorithm explained in the previous sections. All the other individuals are generated randomly. Then a local search is applied on each individual. The details of the local search method(hill climbing) will be explained in the following sections.

## IV.2.3. Ranking Based on Unified Objective metric

The initial population generation phase is followed by the sorting of individuals according to a rank in ascending order. This rank value calculated according to a weighted unified objective for minimizing both makespan and reliability cost and denoted by  $r_u$ . Normalized values are used for makespan and reliability cost by dividing them to population average.

$$\text{rank}_u = w_1 * \text{makespan of } ch_i / \text{avg}_m + w_2 * \text{reliability cost of } ch_i / \text{avg}_{rc} \quad (\text{IV.2})$$

In the above equation  $\text{avg}_m$  denotes the average makespan value of population,  $\text{avg}_{rc}$  denotes average reliability cost value of population,  $ch_i$  denotes  $i$ th chromosome in the population and  $\text{rank}_u$  denotes unified rank.

The population is sorted according to  $r_u$ . The individuals in the front of the population queue are the best ones while the latter ones are the worst.

In each iteration, generated offsprings are inserted into sorted population based on their  $r_u$  values. Because the population size is kept fixed, the individual at the end of the population queue is discarded so that always better individuals are kept in the population.

#### **IV.2.4. Population Model**

In Genetic Algorithms, two types of population models exist: Generational and Steady-State. In the first type, all the individuals are selected to the mating pool which consists of the individuals on which a genetic operator like crossover and mutation will be applied. The generated offsprings take place of their parents and the entire population is replaced. This kind of replacement takes a lot of computational time. Also even the worst offsprings are added to the population. However in steady-state method, the population is changed according to a ratio which is defined by  $\lambda/\mu$  where  $\lambda$  number of individuals are replaced in  $\mu$  number of individuals. Hence good individuals are not discarded and survives in the population. Also the computational time is less for less number of individuals are replaced.

In our algorithm, steady-state approach. In each iteration, two offsprings in crossover, one offspring in mutation phase are generated and inserted into the population. Because the population size is kept fixed all the time, the worst individuals are discarded when new offsprings are inserted.

### IV.2.5. Selection

We use tournament selection method to select parents from a population to apply crossover and mutation operators. The significance of this method is that there is no need for the existing of the global knowledge only we must have some metric to compare two individuals. As our algorithm evaluates individuals according to their rank values, tournament selection is best suited to our needs. The outline of tournament selection is given in the figure IV.4

```
begin  
    Set current_member = 1  
    while(current_member <=population_size)  
        Pick k individuals randomly with or without replacement  
        Select the best of these k comparing their ranks  
        Denote the individual as i  
        Set mating_pool[current_member] = i  
        Set current_member = current_member +1  
    Do  
end
```

Figure IV.4: Tournament Selection[29]

Here, k (tournament size) is a factor that affects the selection. In our algorithm we use 2 and 3 for tournament size. For k=3, three parents are selected randomly, two of them are chosen according to their unified rank based on makespan and reliability cost factors. Then a crossover operator is applied. The same method is used for the selection of one parent to apply mutation operator. The greater the size, it is more likely that better individuals are selected.

### IV.2.6. Local Search (Hill Climbing)

The local search hill climbing is a method that takes a certain individual and moves to a neighbouring solution if it is better than the current one. This operation

continues until a stopping condition is met. The local search can be applied in many phases of a genetic algorithm (see Figure IV.5).

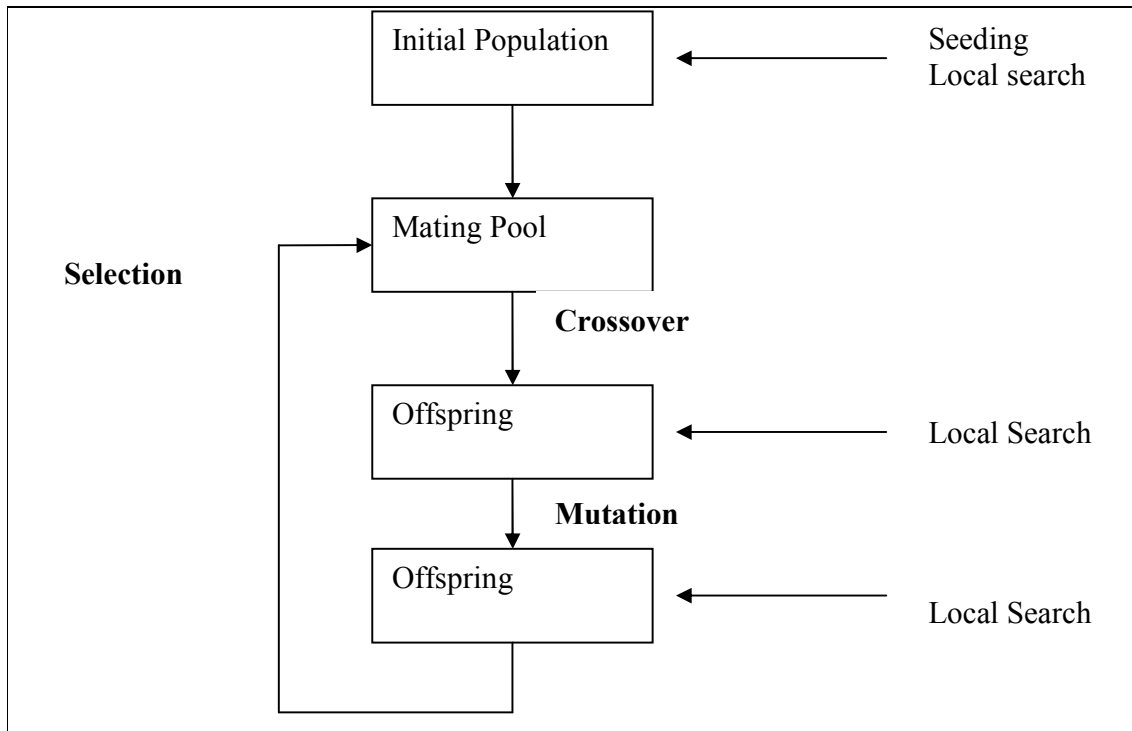


Figure IV.5: Application areas of local search in a hybrid GA[29]

In our algorithm local search is applied on each individual in initial population, crossover and mutation phase. Two tasks are randomly selected from the chromosome and they are replaced. As a second approach only the processor assigned to a randomly selected task is changed. The rank of the resulting offspring is calculated. If the offspring has a better rank than the older one is discarded. This goes on for 500 iterations. Then the offspring is inserted to the population based on its rank. The hill climbing is also applied to the offsprings that are generated after the crossover and mutation operations.

### IV.2.7. Crossover

We consider two different crossover operators in our Memetic Algorithm: a) one point order crossover, b) uniform crossover. While in the first one the order of tasks changes, in the latter one only processor assignments are changed. In One Point Order

Crossover, the selected two parents are cut at a randomly selected point. The left of the first parent becomes the first part of the first offspring. The rest of the offspring genes are the ones that consists of tasks that does not exist in the left part of the first offspring and in an order that is in the second parent with its processor assignments. The order of the tasks is preserved which is also beneficial for our arrival time constraint.

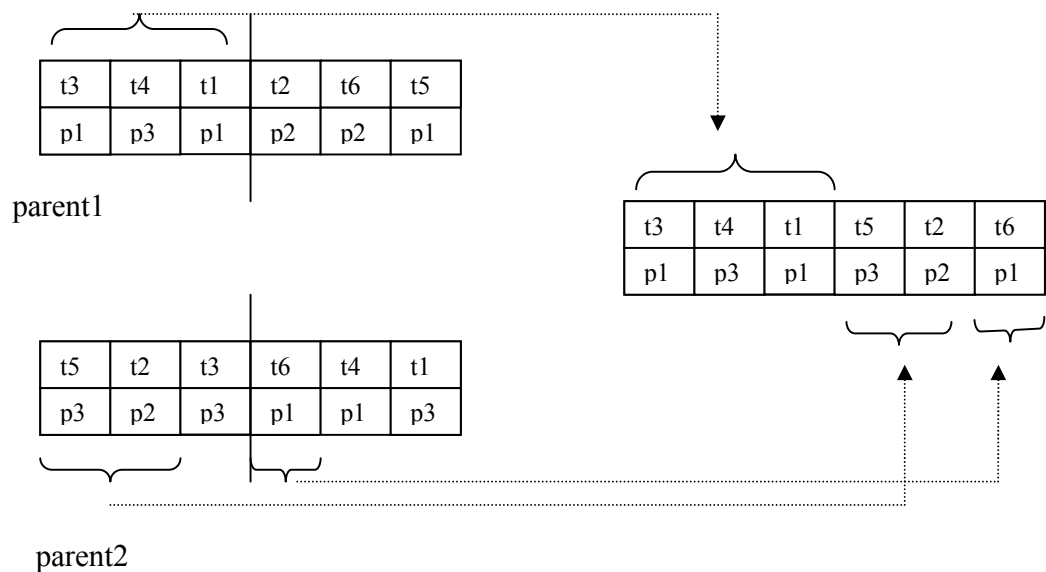


Figure IV.6: One point order crossover

An example is given in Figure IV.6 which shows the first offspring. The second offspring is generated using the same method but this time based on the second parent. In the figure the left part of parent1  $\{(t_3, p_1), (t_4, p_3), (t_1, p_1)\}$  becomes the left part of offspring 1. The remaining genes of parent 2  $\{(t_5, p_3), (t_2, p_2), (t_6, p_1)\}$  becomes the remaining part the offspring. The second offspring generated is  $\{(t_5, p_3), (t_2, p_2), (t_3, p_3), (t_4, p_3), (t_1, p_1), (t_6, p_2)\}$ .

In uniform crossover, a series of random numbers in range  $[0,1]$  is generated according to uniform distribution. This number is equal to the chromosome size. If the  $i$ th number generated is less than a predefined parameter  $p$ , the processor number of the  $i$ th gene comes from the processor number of the  $i$ th gene of parent 1, otherwise it comes from parent 2. Usually this parameter is set to 0.5. The uniform crossover operates on genes independently which makes it a suitable operator for independent

task scheduling problem. An example is given in figure IV.9. The random number string used is (0.31,0.67, 0.12, 0.45, 0.92, 0.75).

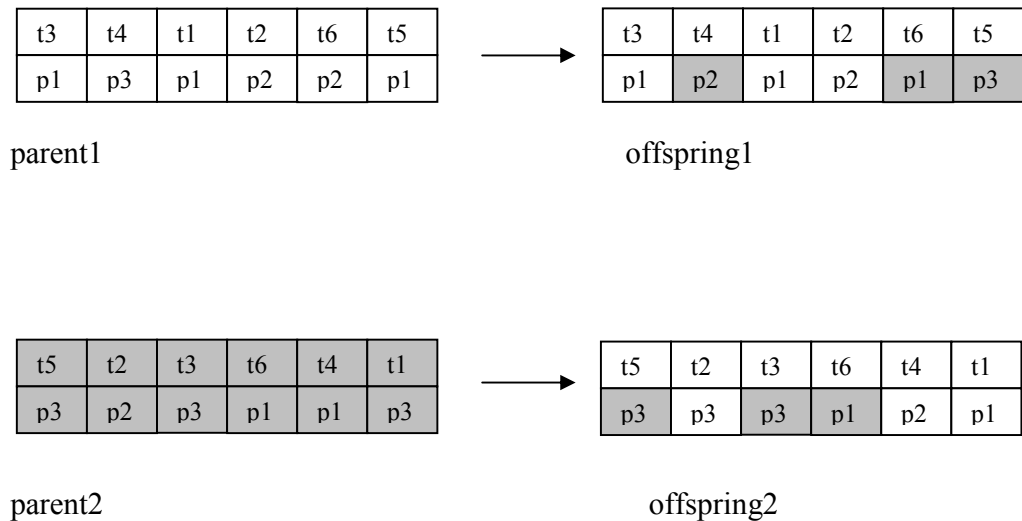


Figure IV.7: Uniform Crossover

As it could be seen from the figure task execution orders does not change but instead processor numbers are changed. For the values 0.31, 0.12, 0.45, both parent genes do not change but for the other values they change their processor assignment to the other parent's.

### IV.2.8. Mutation

There are three different mutation operators considered[26] in our algorithm. The first one is a swap mutation. A parent is selected from the mating pool. On that parent two task-processor pairs are selected randomly and their positions in the chromosome are swapped. In other words in here task execution orders change but their processor assignments stay the same.

The second type of mutation does not change the order. Instead it selects a task from the chosen parent randomly and assign it to another processor which is also

randomly selected. Another name for this type of mutation is rescheduling or reassignment.

The third type of mutation makes use of both first and second type of mutation. It selects two tasks, swap them and change their processor assignment. The figure below shows the application of three types of mutation.

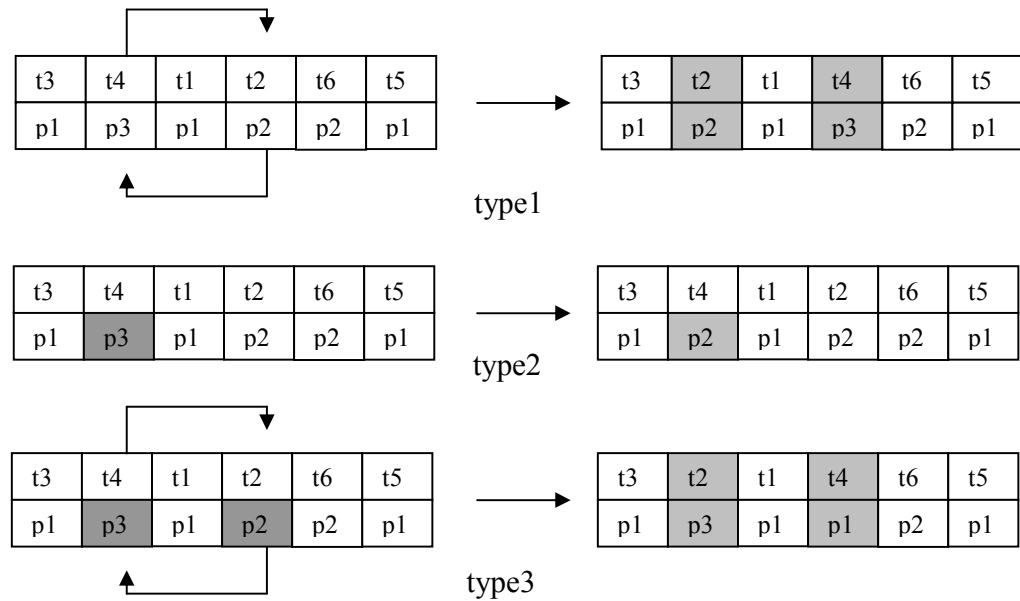


Figure IV.8: Mutation operators

In the figure, for type 1 mutation genes (t4,p3) and (t2,p2) are swapped. For type2 mutation . For type2 mutation the processor assignment of gene (t4,p3) is changed to (t4,p2). In the type3 mutation, first, genes (t4,p3) and (t2,p2) are swapped then gene(t4,p3) is changed to (t4,p1) while gene (t2,p2) is changed to (t2,p3).

## IV.2.9. Outline of the algorithm

The algorithm starts with an initial population generation. Then a local search is applied on this population. The individuals are sorted according to their unified rank. For a number of iterations crossover and mutation operations followed by local search are applied on individuals which are selected with tournament selection. New individuals are inserted into population according to their unified rank. The population size is kept fixed at 200.

- (1) Generate initial population
- (2) Perform local search(hill climbing) on each individual of the population
  - While**(stopping condition is not met)
    - Select two tasks randomly
    - Change their processor assignment (or order)
    - if** unified rank of offspring is greater than the original
      - replace the original with the offspring otherwise discard it
  - end while
- (3) Evaluate unified ranks for each individual
- (4) Sort them according to their ranks
- (5) **While**(stopping condition is not met)
  - (6) Select two parent (tournament selection for k in [2,3])
  - (7) Perform crossover
  - (8) Perform hill climbing on the offsprings
  - (9) Insert them to the population according to their unified rank value
  - (10) Select one parent as in step 6.
  - (11) Perform mutation
  - (12) Perform hillclimbing on the offspring
  - (13) Insert it to the population according to its unified rank value
  - (14) Keep population size fixed.
- (15) **end while**
- (16) Output the best solution.

Figure IV.9: Memetic Algorithm with a unified objective

# **PART V**

## **EXPERIMENTAL STUDY**

This section presents the details of experimental evaluation of our algorithms by comparing their performance with leading algorithms in the literature.

In order to provide comparisons, set of independent tasks with different characteristics are generated. Additionally the number of machines and machine failure rates are required to set the test cases. The range of values for the number of machines used in our study is [20,40,60,80]. The failure rates of machines are generated according to Poisson distribution with a mean value of  $2.5 * 10^{-5}$ .

### **V.1. PERFORMANCE EVALUATION OF IMMEDIATE MODE ALGORITHMS**

In addition to previously mentioned parameters(which are number of tasks, task & machine heterogeneity, number of machines), arrival rates of tasks must also be given. The interarrival times are calculated based on mean execution time and a parameter  $\alpha$  which is a coefficient to increase or decrease interarrival time. The range of values for  $\alpha$  is [0.1,0.25,0.5,0.75]. As the value of  $\alpha$  increases the interarrival times increase too. Thus a small value of  $\alpha$  means bursty arrival rate and does not affect makespan much while a large value does.

The reliability based algorithm that we proposed also needs weight values  $w_1$  and  $w_2$  to construct the unified objective from makespan and reliability cost. The value sets are  $\{[0,1],[0.25,0.75],[0.5,0.5],[0.75,0.25],[1,0]\}$  respectively. The value of weights specifies the importance of objective in the execution of the algorithm.

Three performance metrics considered are makespan, reliability cost and unified objective of the previous two. The calculation of makespan and reliability cost was explained in previous sections. The unified objective is calculated with normalized values of makespan and reliability cost multiplied by their weights. Normalization is applied by the following formula.

$$NormalizedMakespan = (makespan - makespan_{min}) / (makespan_{max} - makespan_{min})$$

The same formula is valid for the normalization of reliability costs except makespan is replaced with reliability cost.

### V.1.1. Results and Discussion

In this section, test case combinations are given and their results are discussed for evaluation. There are four test cases used to measure the performance metrics.

The first test case mainly is aimed to present the effect of number of tasks on the resultant performance metrics for different combinations of task machine heterogeneity and unified objective weights. 50 different task graphs are tested and their average result is taken. The processor number is kept at 40 and  $\alpha$  is kept at 0.25. Four different heterogeneity cases are tested, which are LoLo, LoHi, HiLo, HiHi. Number of tasks change between 100 and 1000. Three different weight combinations are applied:  $W_1 = \{w_1, w_2\} = \{0.25, 0.75\}$ ,  $W_2 = \{w_1, w_2\} = \{0.5, 0.5\}$ ,  $W_3 = \{w_1, w_2\} = \{0.75, 0.25\}$ .

For the LoLo heterogeneity case and  $W_1$  is applied, RKPb gives the worst makespan value. MET, MCT, SA and KPb follows it. In contrast RKPb gives the least reliability cost value and outperforms the others. For unified objective, again RKPb outperforms the others and gives better results. When the weights are equal RKPb

outperforms MET and gets closer to the other three algorithm in terms of makespan. It outperforms all other algorithms in terms of reliability cost. For the unified objective RKPb outperforms all others, as task size increases the curves of MET and RKPb get closer to each other. When  $W_3$  is applied ,RKPb outperforms MET and as task size increases it outperforms MCT, SA and KPb in terms of makespan. But in reliability cost MET gives better results than RKPb. In unified objective RKPb outperforms the others but as number of tasks increases MET outperforms it. When the task and machine heterogeneity increases the results become more deterministic. In unified objective metric, RKPb outperforms all algorithms in every case except when  $W_3$  is applied. In  $W_3$ , MET outperforms it in terms of reliability cost and unified objective. The reason why MET gives good results for reliability cost is that MET chooses least execution times and this results in least values when multiplied with failure rate.

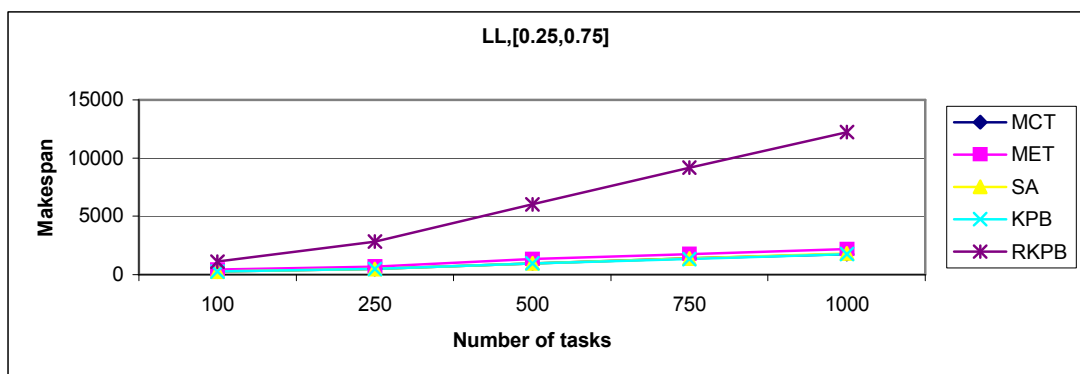


Figure V.1. Makespan vs. Number of tasks, LoLo heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

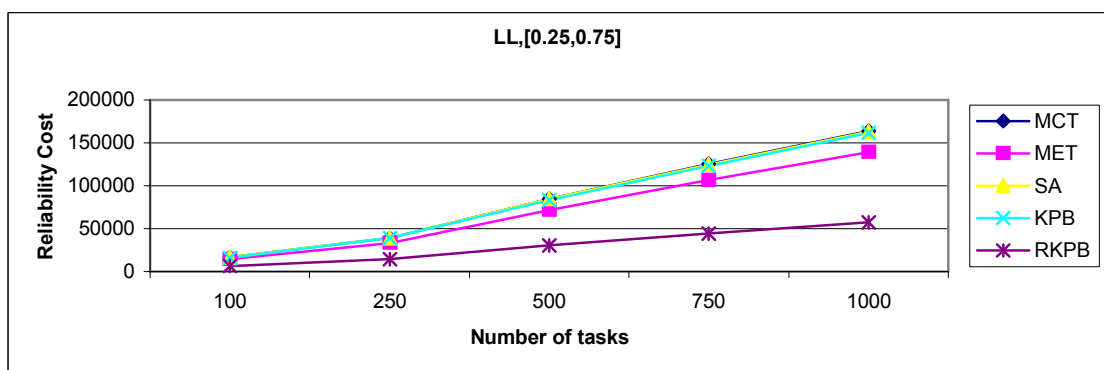


Figure V.2. Reliability Cost vs. Number of tasks, LoLo heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

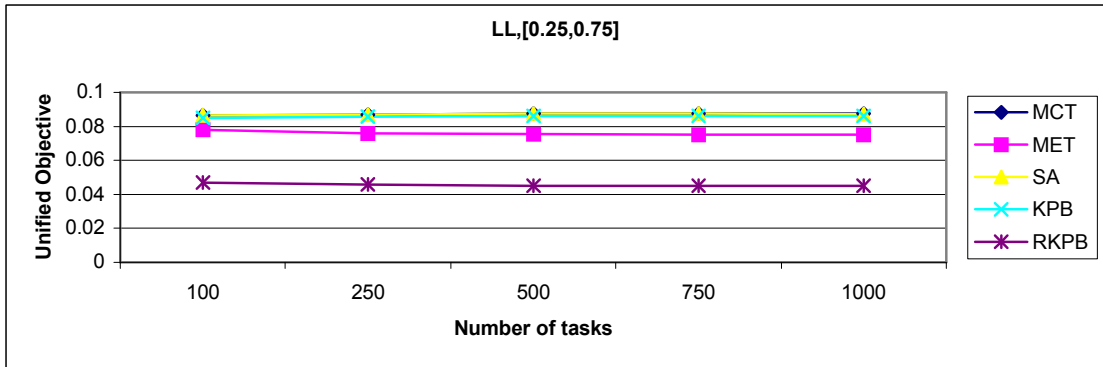


Figure V.3. Unified Objective vs. Number of tasks, LoLo heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

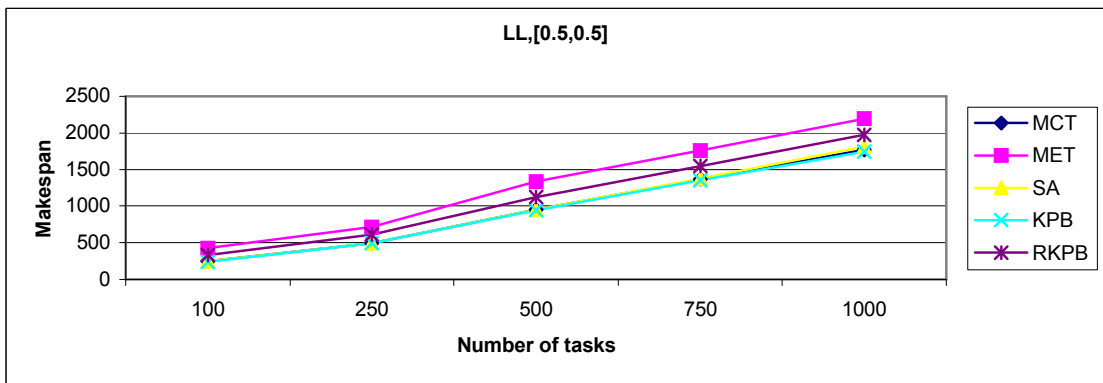


Figure V.4. Makespan vs. Number of tasks, LoLo heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

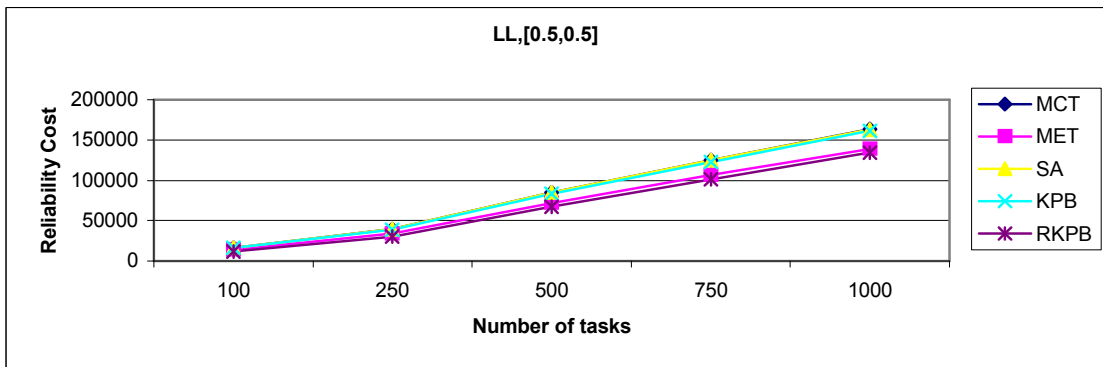


Figure V.5. Reliability Cost vs. Number of tasks, LoLo heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

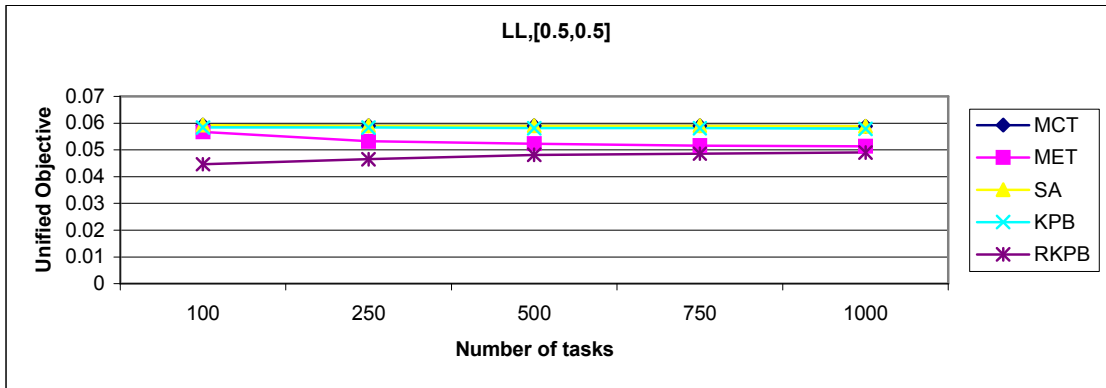


Figure V.6. Unified Objective vs. Number of tasks, LoLo heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

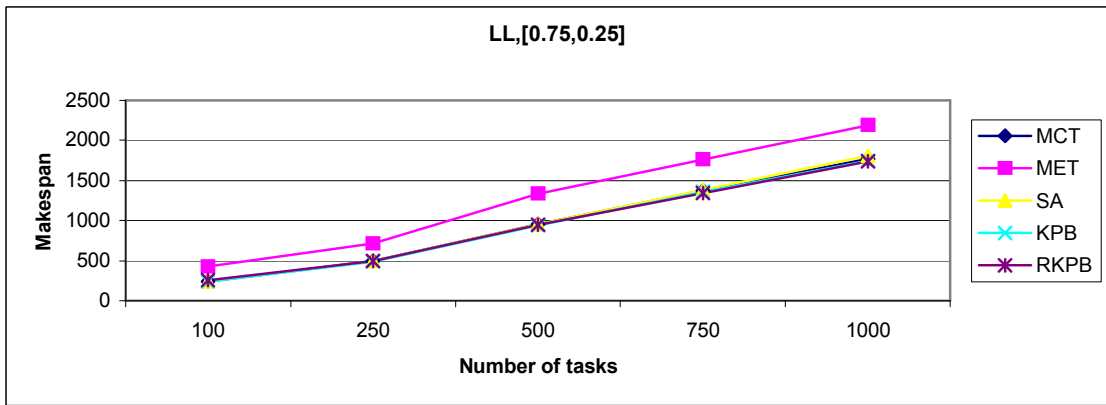


Figure V.7. Makespan vs. Number of tasks, LoLo heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

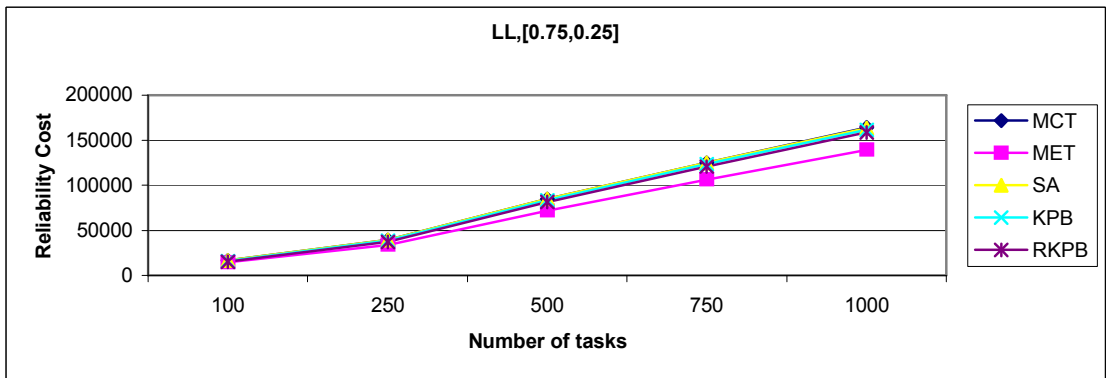


Figure V.8. Reliability Cost vs. Number of tasks, LoLo heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

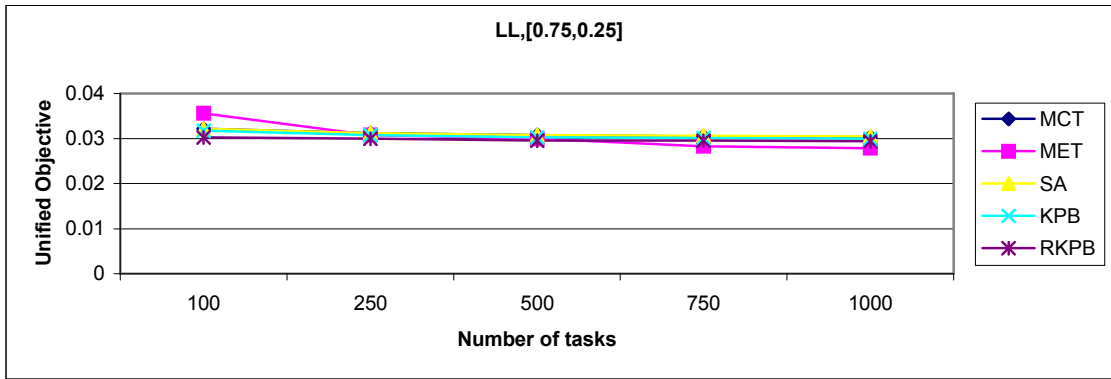


Figure V.9. Unified Objective vs. Number of tasks, LoLo heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

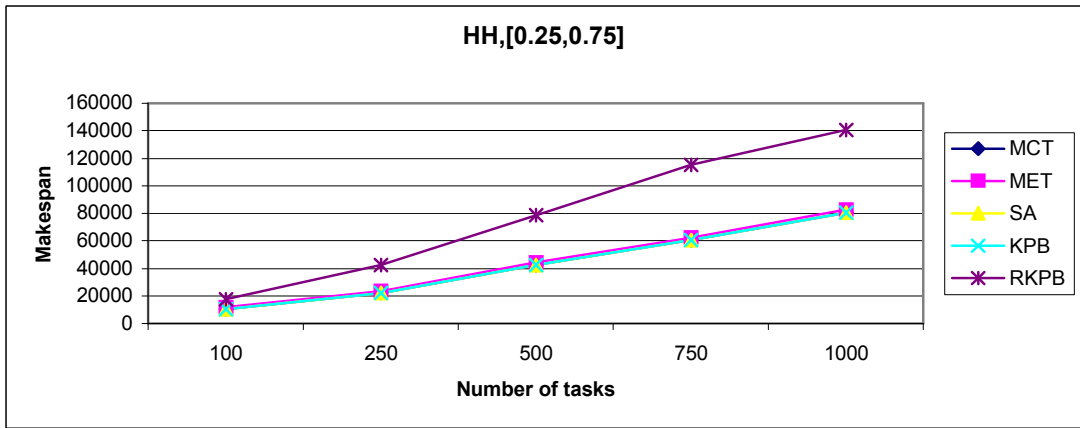


Figure V.10. Makespan vs. Number of tasks, HiHi heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

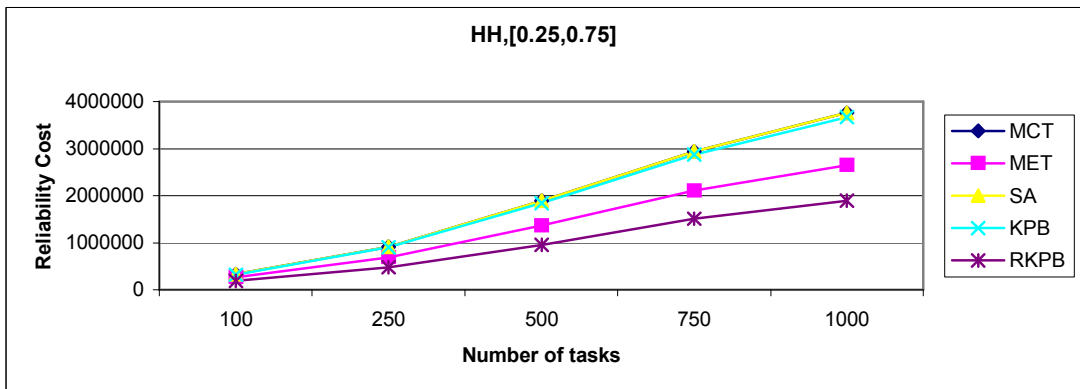


Figure V.11. Reliability Cost vs. Number of tasks, HiHi heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

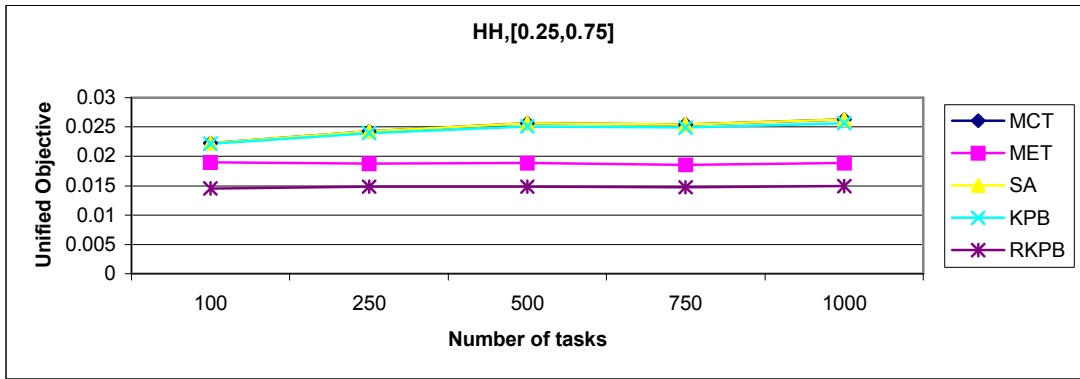


Figure V.12.Unified Objective vs. Number of tasks, HiHi heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

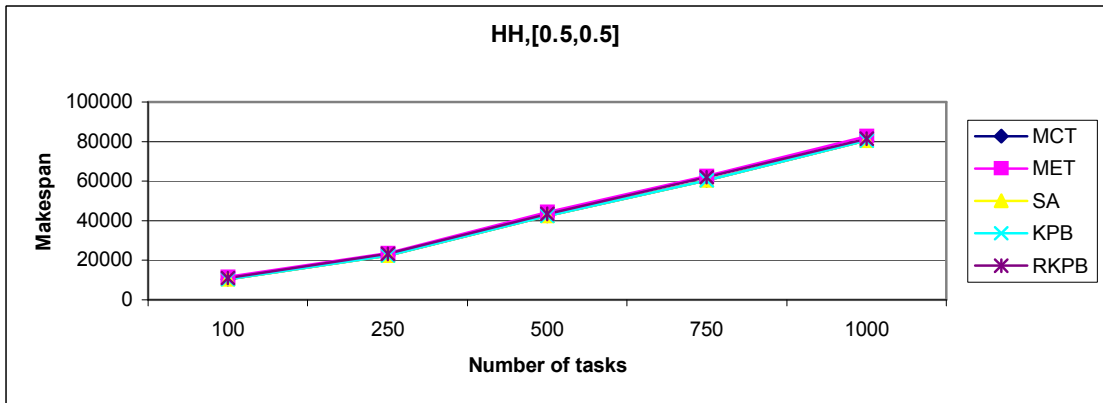


Figure V.13.Makespan vs. Number of tasks, HiHi heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

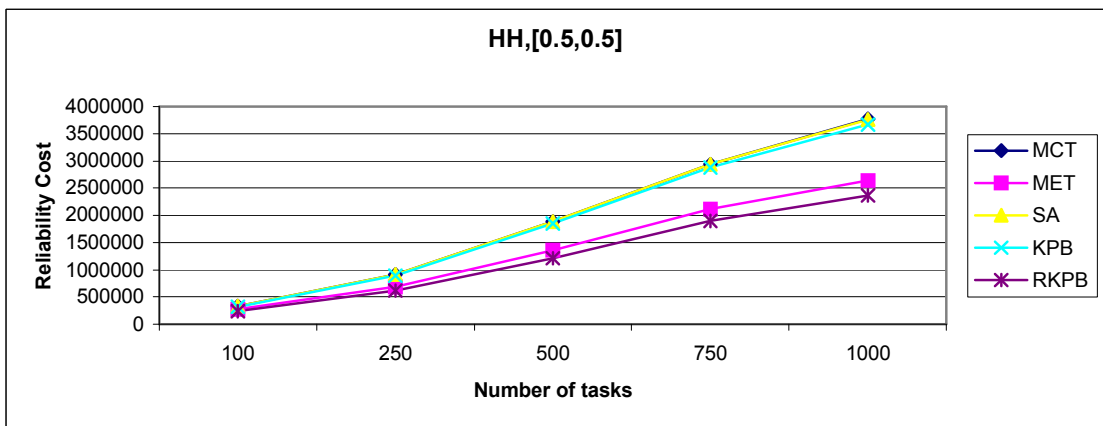


Figure V.14.Reliability Cost vs. Number of tasks, HiHi heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

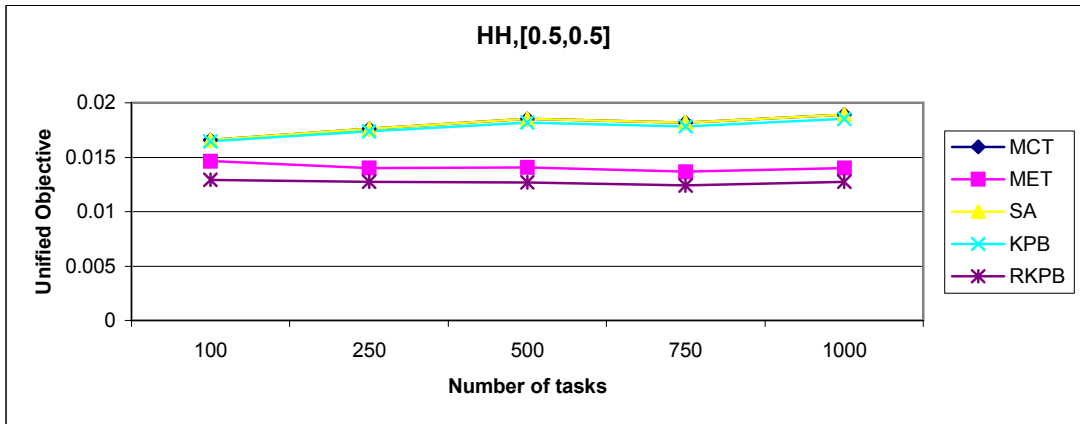


Figure V.15.Unified Objective vs. Number of tasks, HiHi heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

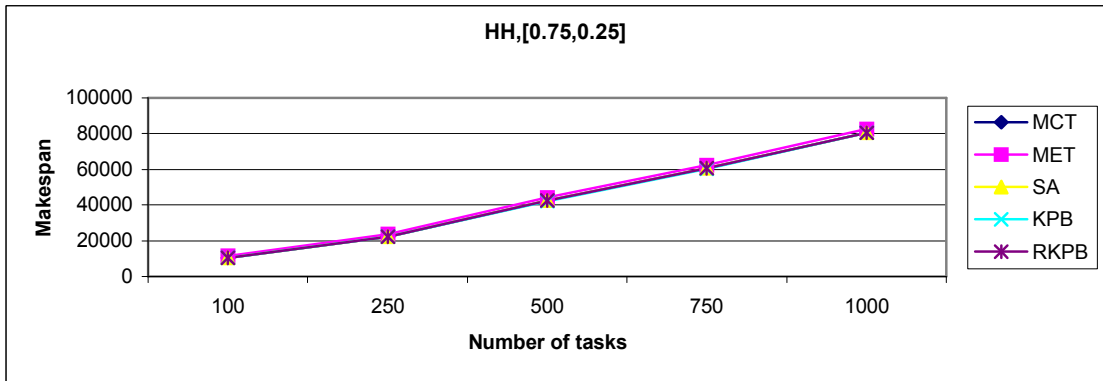


Figure V.16.Makespan vs. Number of tasks, HiHi heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

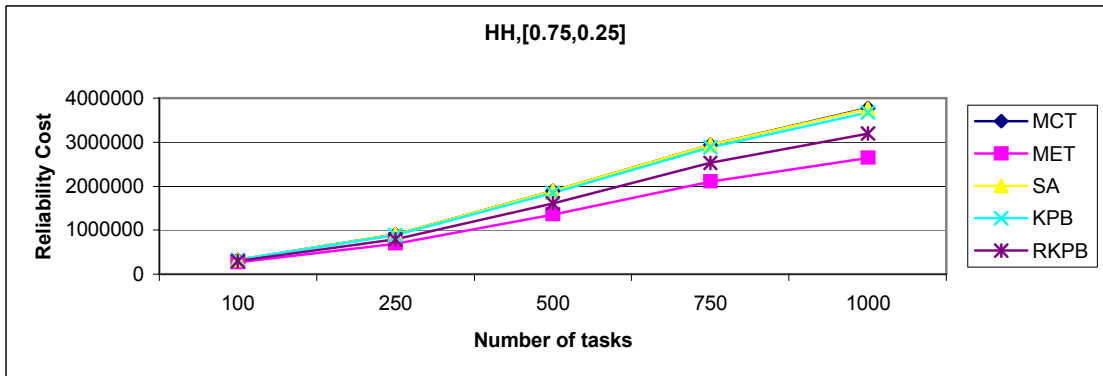


Figure V.17.Reliability Cost vs. Number of tasks, HiHi heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

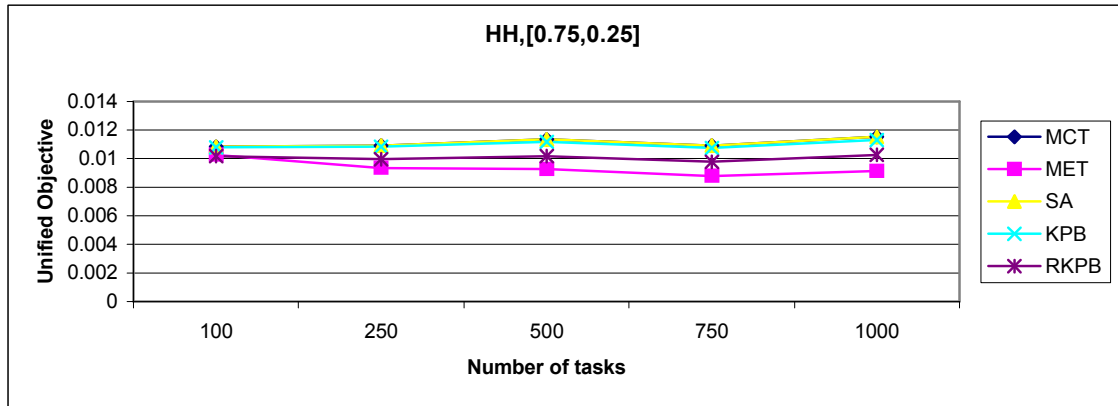


Figure V.18. Makespan vs. Number of tasks, HiHi heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

In the second test case the effect of arrival rates are measured. The processor number is held constant at 20 and number of tasks at 500. In this case only LoLo and HiHi task-machine heterogeneity cases are considered. As arrival rate increases, makespan also increases but this increase can be seen more clearly for the HiHi heterogeneity case. The reason for this is that interarrival times are calculated based on a formula that includes mean execution time of tasks. As the heterogeneity values increase, mean execution time also increases and results in a higher makespan. The reliability cost does not show a definite characteristic based on arrival rate. For the LoLo heterogeneity case, RKPB outperforms all other algorithms when  $W_1$  applied, outperforms MET for  $W_2$  in terms of makespan. For HiHi heterogeneity case RKPB gives very close results in terms of makespan especially when the arrival rate is increased when  $W_1$  is applied thus the gap among the other algorithms is decreased in terms of makespan. The reason for this is that processors become idle in each case so that both RKPB and other algorithms gives longer makespan values. For  $W_2$ , it outperforms MET in makespan and all other algorithms when  $W_3$  is applied and the arrival rate is small. In all cases except for  $W_3$  it outperforms other algorithms in terms of reliability cost and unified objective. For  $W_3$ , MET outperforms RKPB in reliability cost and unified objective.

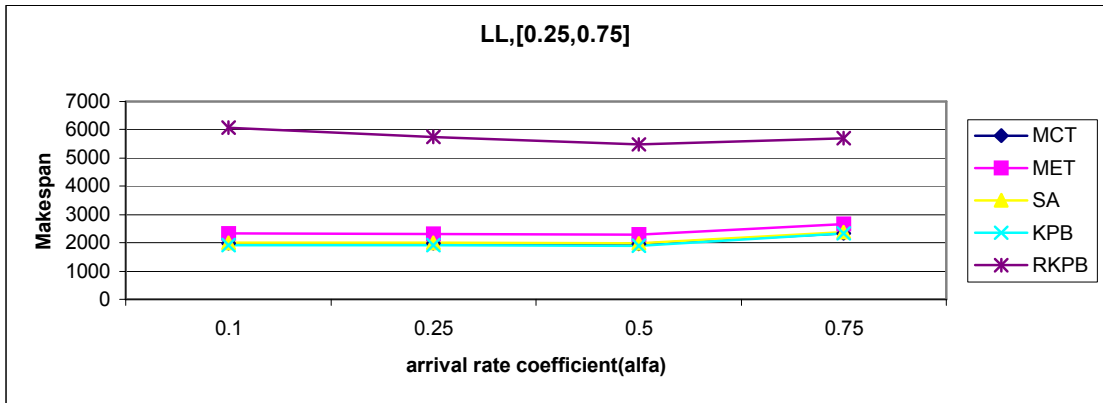


Figure V.19. Makespan vs.  $\alpha$ , LoLo heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

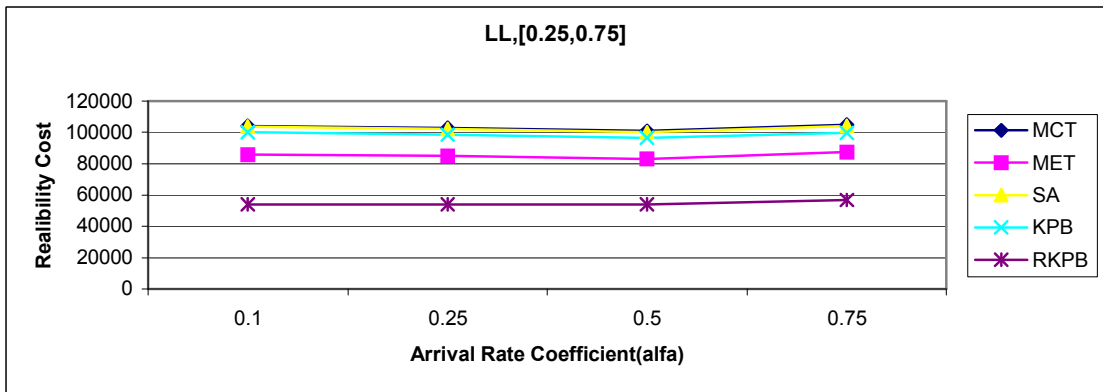


Figure V.20. Reliability Cost vs.  $\alpha$ , LoLo heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

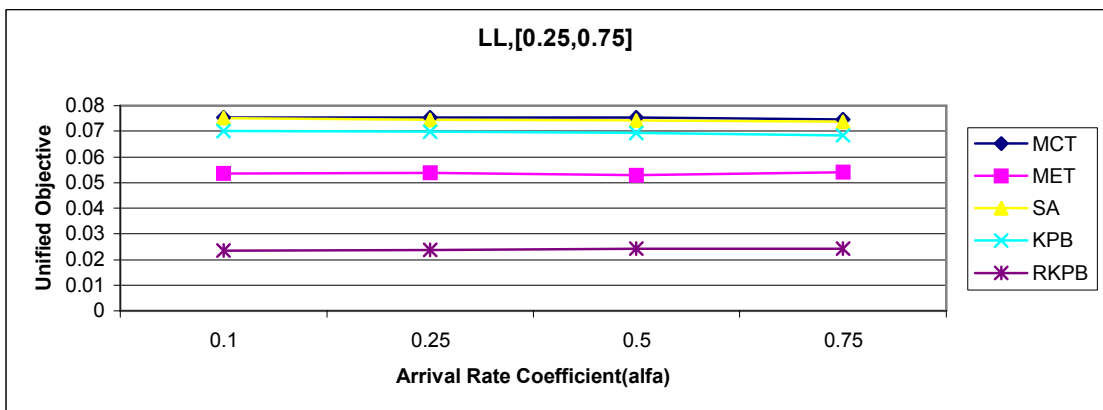


Figure V.21. Unified Objective vs.  $\alpha$ , LoLo heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

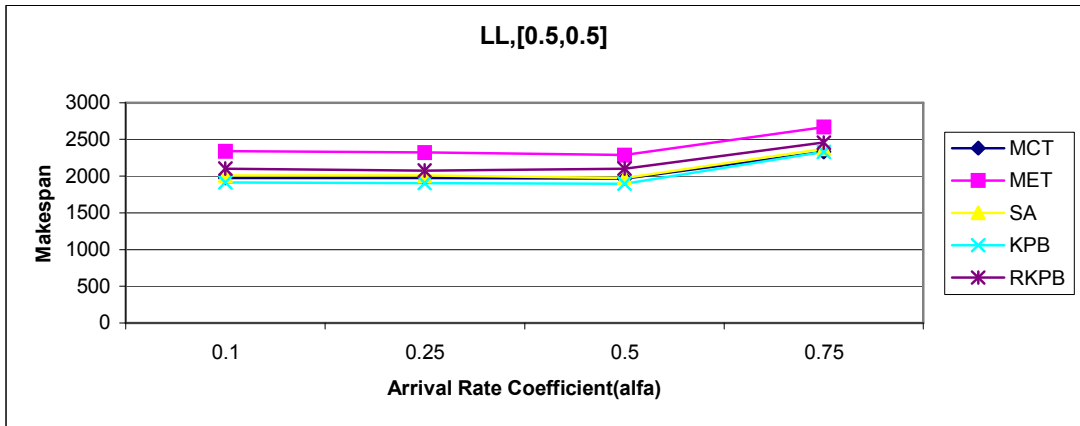


Figure V.22. Makespan vs.  $\alpha$ , LoLo heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

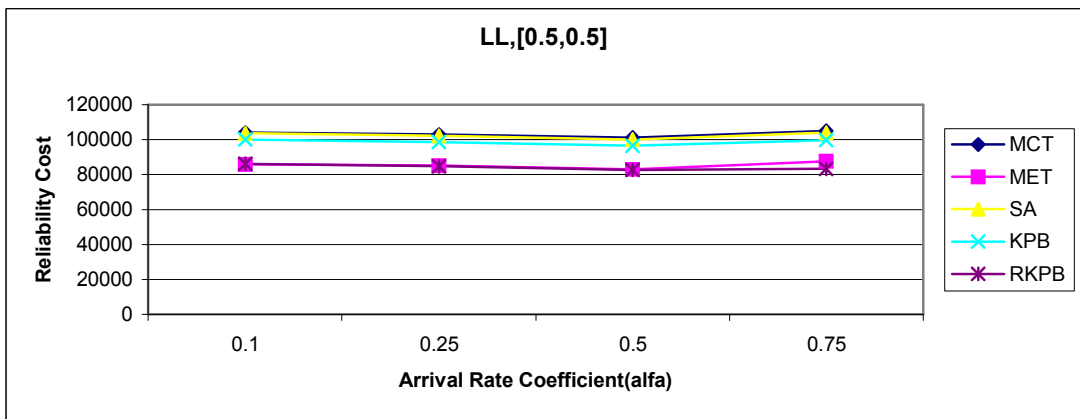


Figure V.23. Reliability Cost vs.  $\alpha$ , LoLo heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

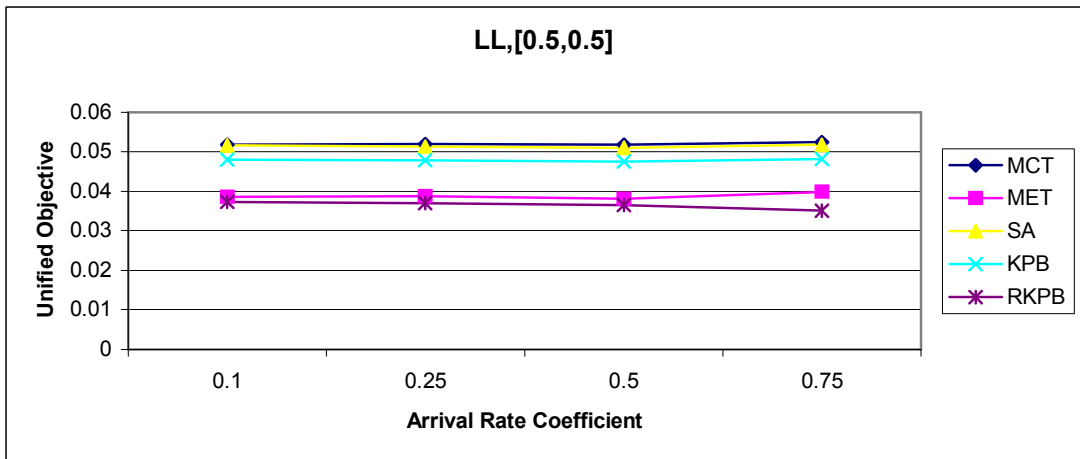


Figure V.24. Unified Objective vs.  $\alpha$ , LoLo heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

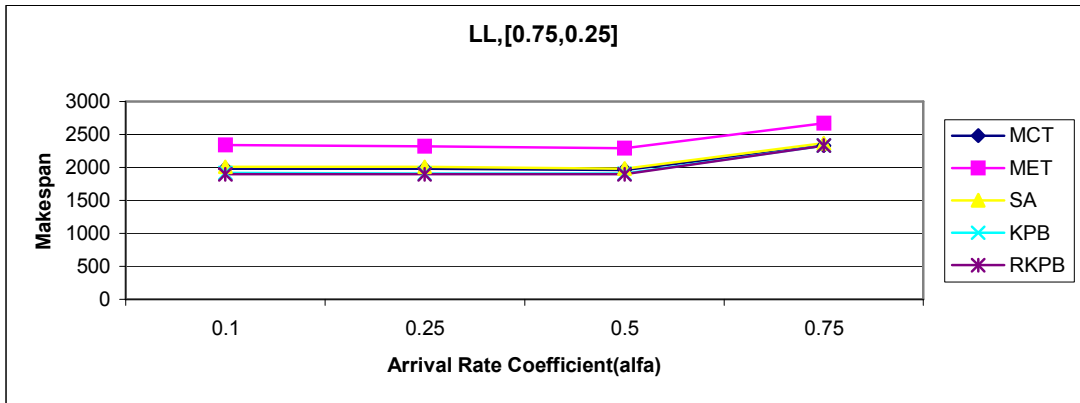


Figure V.25. Makespan vs.  $\alpha$ , LoLo heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

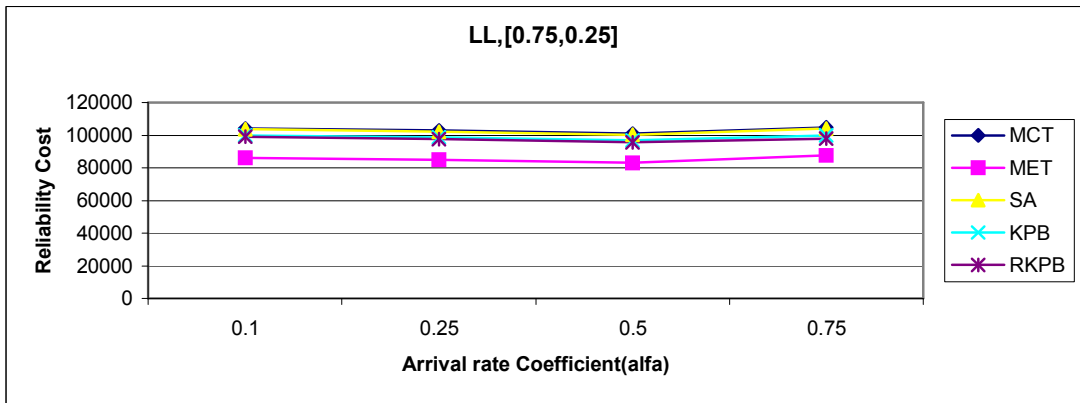


Figure V.26. Reliability Cost vs.  $\alpha$ , LoLo heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

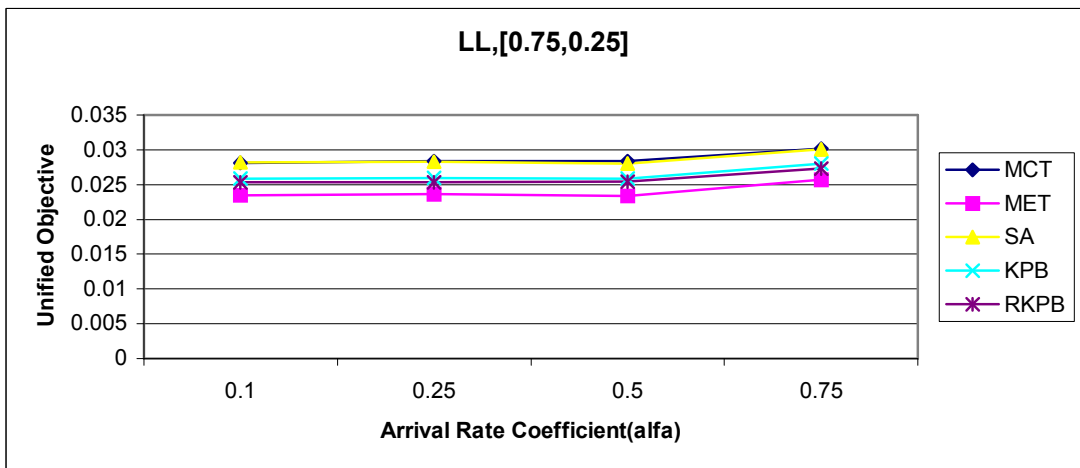


Figure V.27. Unified Objective vs.  $\alpha$ , LoLo heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

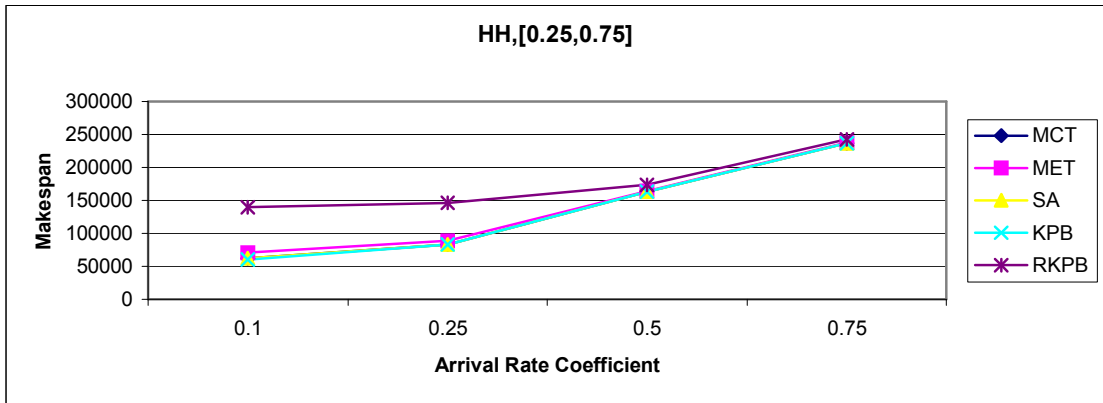


Figure V.28. Makespan vs.  $\alpha$ , HiHi heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

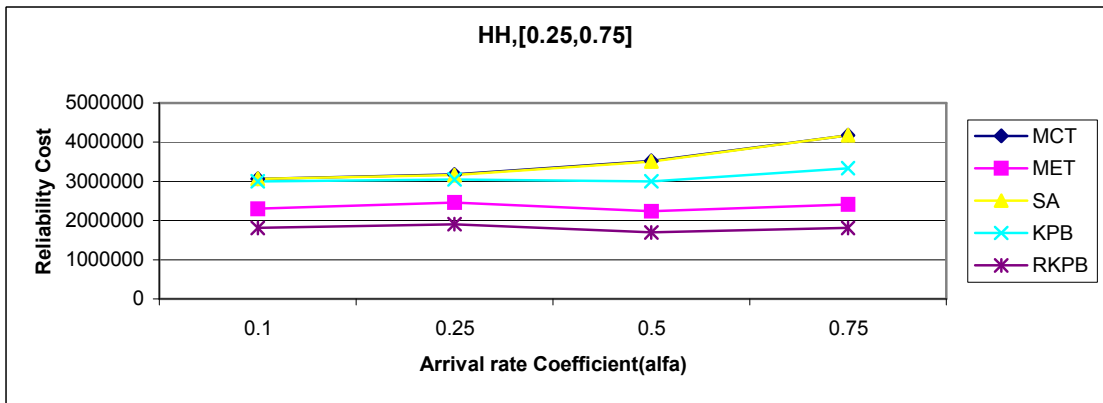


Figure V.29. Reliability Cost vs.  $\alpha$ , HiHi heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

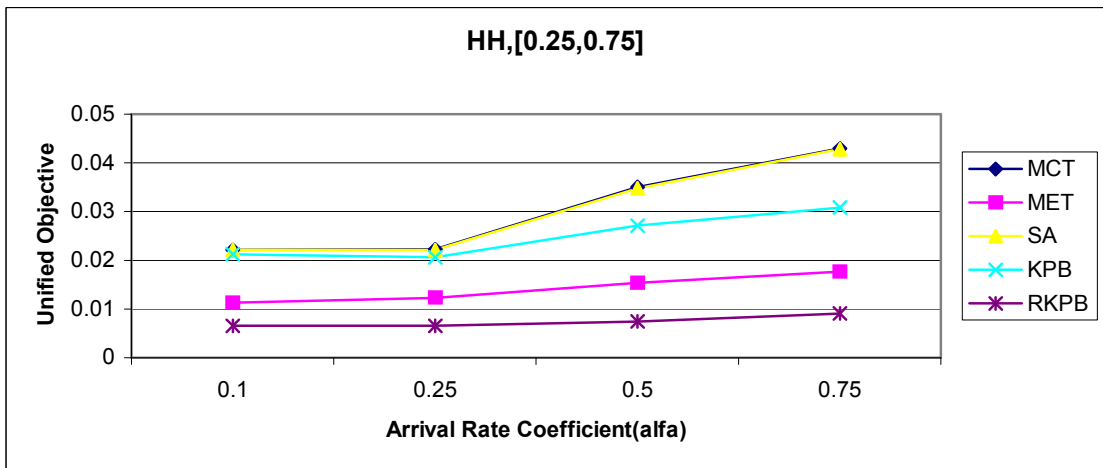


Figure V.30. Unified Objective vs.  $\alpha$ , HiHi heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

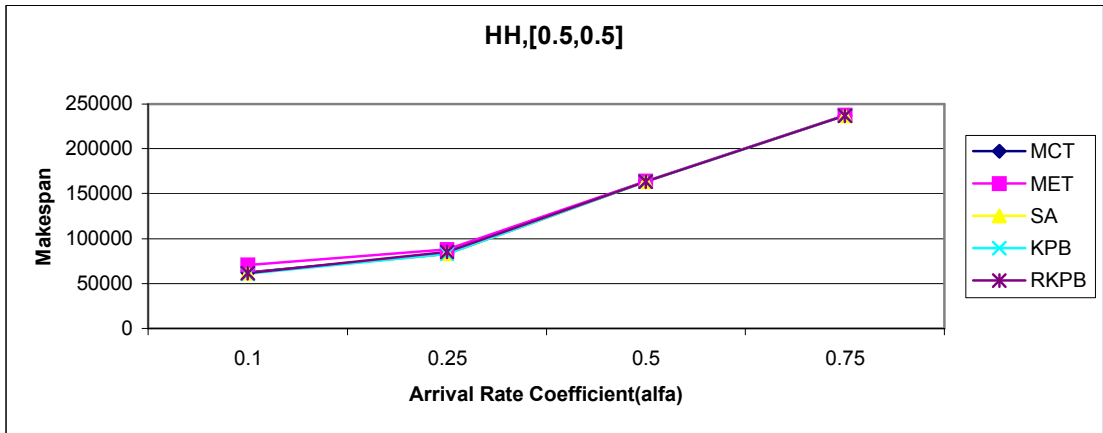


Figure V.31. Makespan vs.  $\alpha$ , HiHi heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

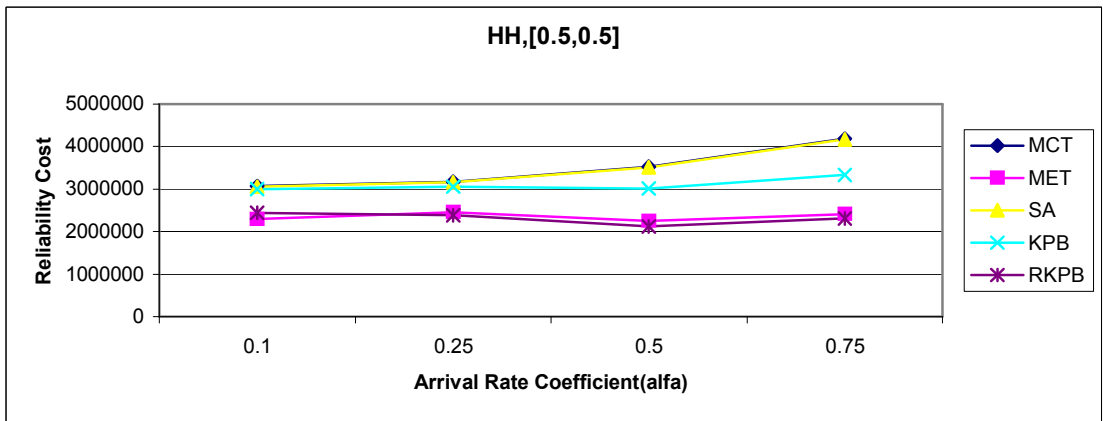


Figure V.32. Reliability Cost vs.  $\alpha$ , HiHi heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

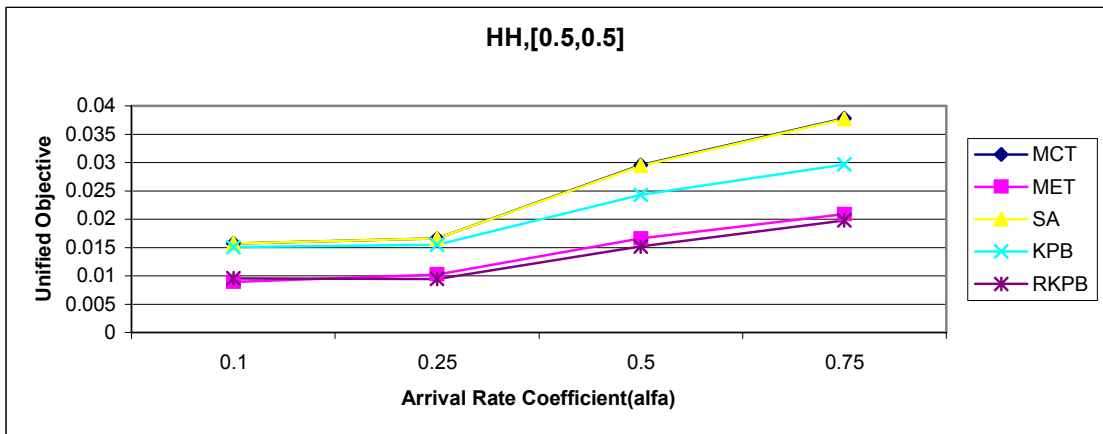


Figure V.33. Unified Objective vs.  $\alpha$ , HiHi heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

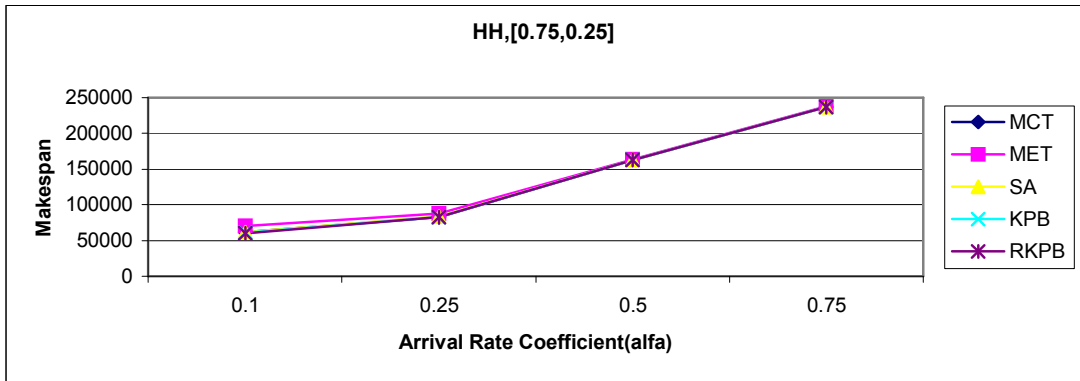


Figure V.34. Makespan vs.  $\alpha$ , HiHi heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

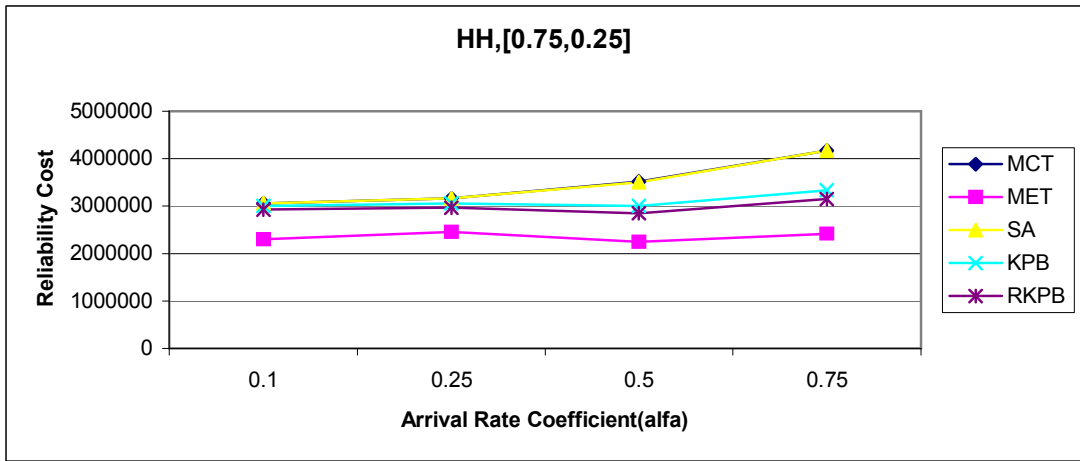


Figure V.35. Reliability Cost vs.  $\alpha$ , HiHi heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

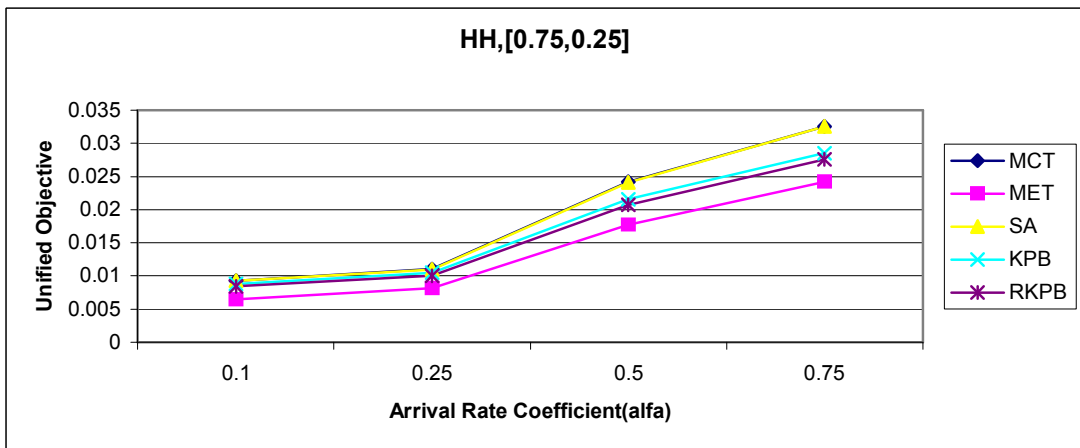


Figure V.36. Unified Objective vs.  $\alpha$ , HiHi heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

In the third test case the effect of number of processors are measured. Arrival rates are held constant at 0.25, number of tasks at 1000. Only LoLo and HiHi heterogeneity cases are considered. For LoLo case when  $W_2$  is applied RKPB outperforms MET and gives very close results to other algorithms in makespan but for  $W_3$ , RKPB outperforms MCT, MET, SA when number of processors is small. When HiHi heterogeneity is applied however RKPB gives so close results in terms of makespan to the other algorithms when processor number is high even for  $W_1$ . For  $W_2$  and  $W_3$  it gives good results independent of the processor number. For every case except for  $W_3$ , RKPB outperforms all other algorithms in terms of reliability cost and unified objective. When  $W_3$  is applied MET outperforms RKPB. In general makespan and reliability cost tend to decrease when the number of processors are increased.

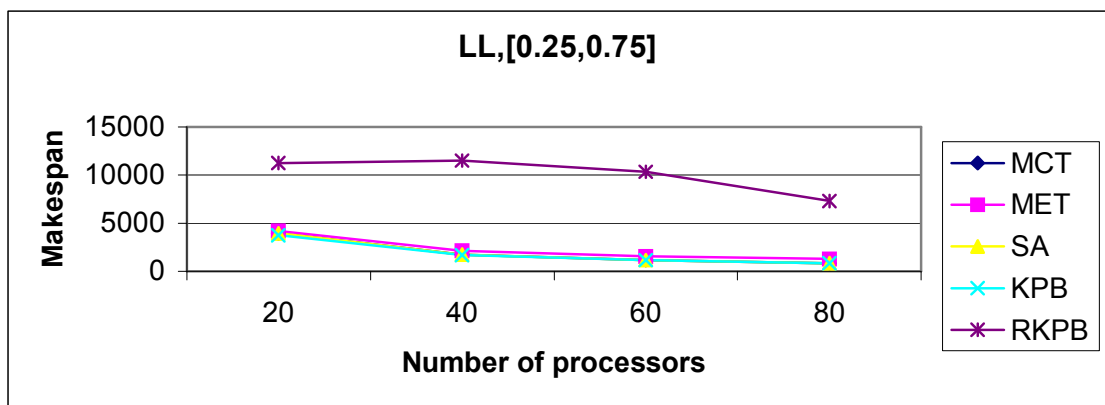


Figure V.37. Makespan vs. Number of processors, LoLo heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

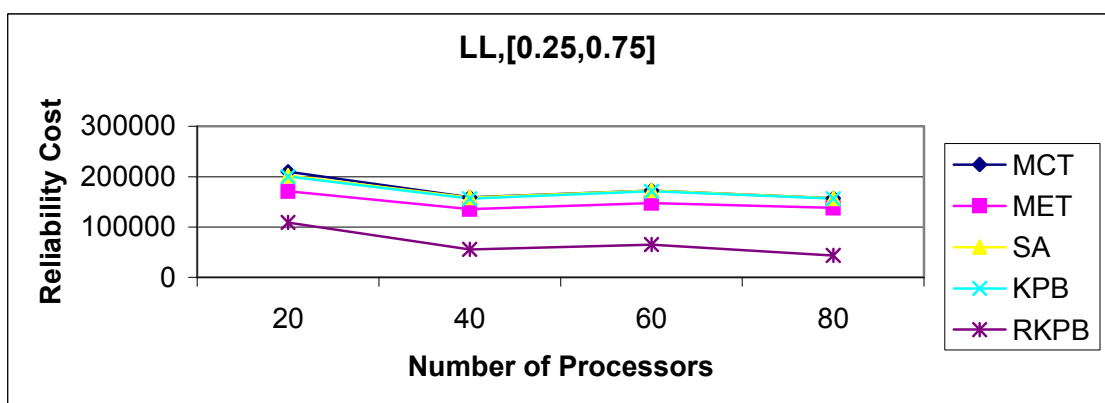


Figure V.38. Reliability Cost vs. Number of processors, LoLo heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

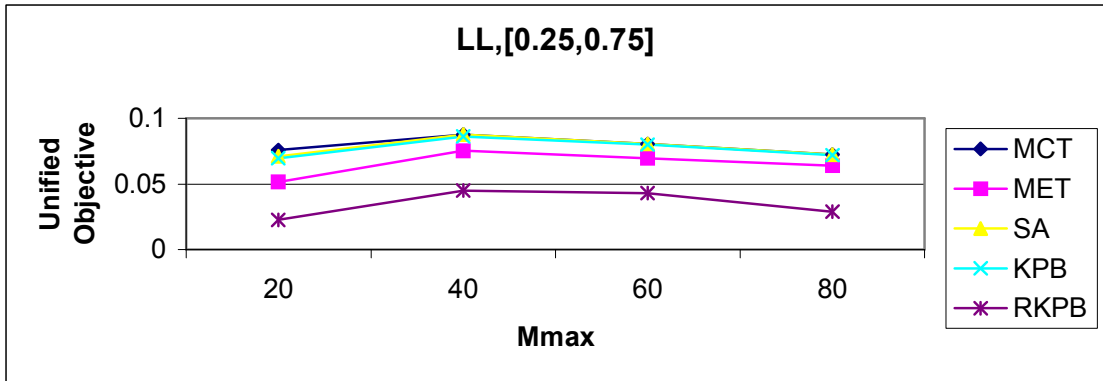


Figure V.39. Unified Objective vs. Number of processors, LoLo heterogeneity,  $w_1=0.25, w_2=0.75$

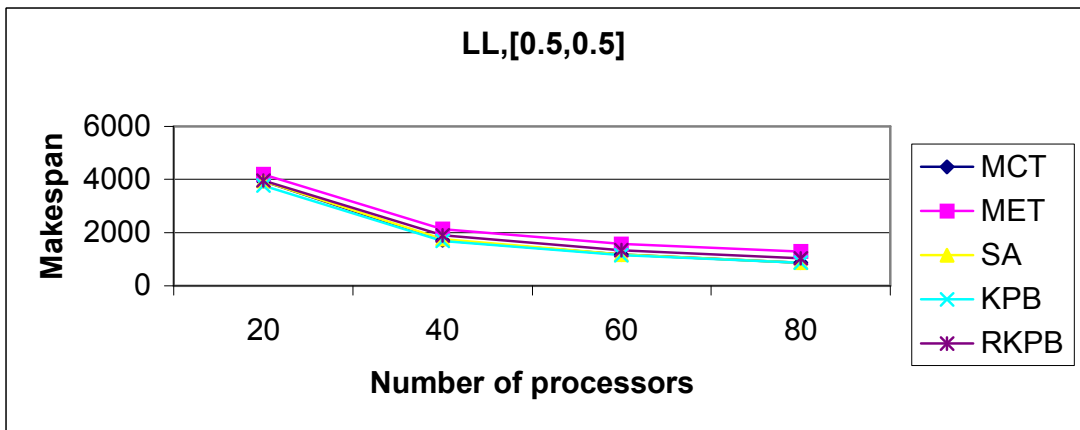


Figure V.40. Makespan vs. Number of processors, LoLo heterogeneity,  $w_1=0.5, w_2=0.5$

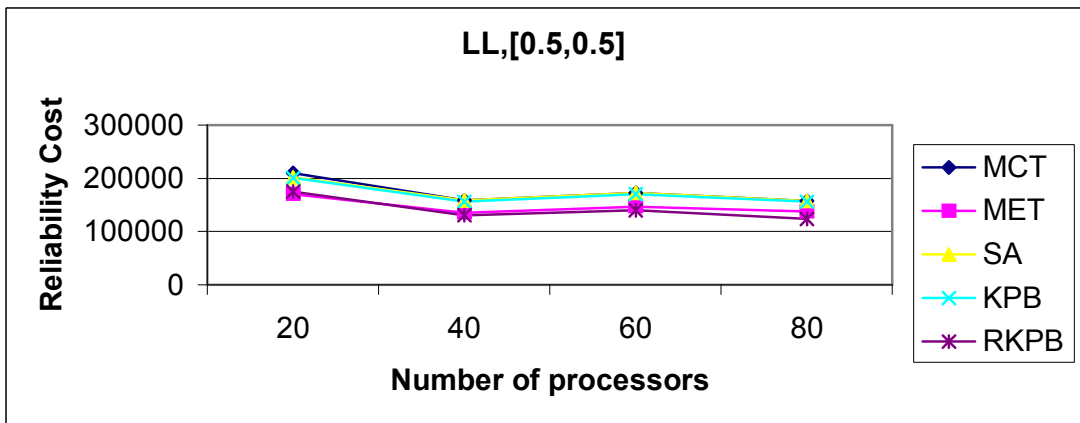


Figure V.41. Reliability Cost vs. Number of processors, LoLo heterogeneity,  $w_1=0.5, w_2=0.5$

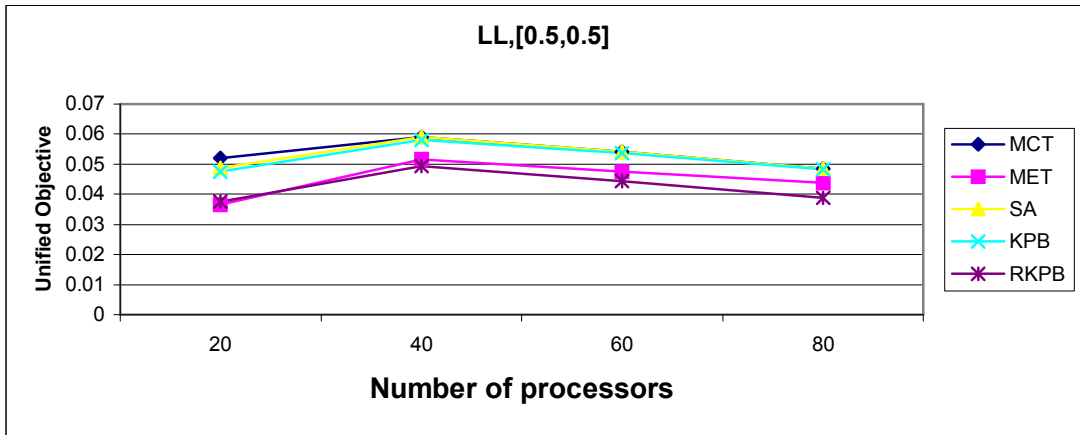


Figure V.42. Unified Objective vs. Number of processors, LoLo heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

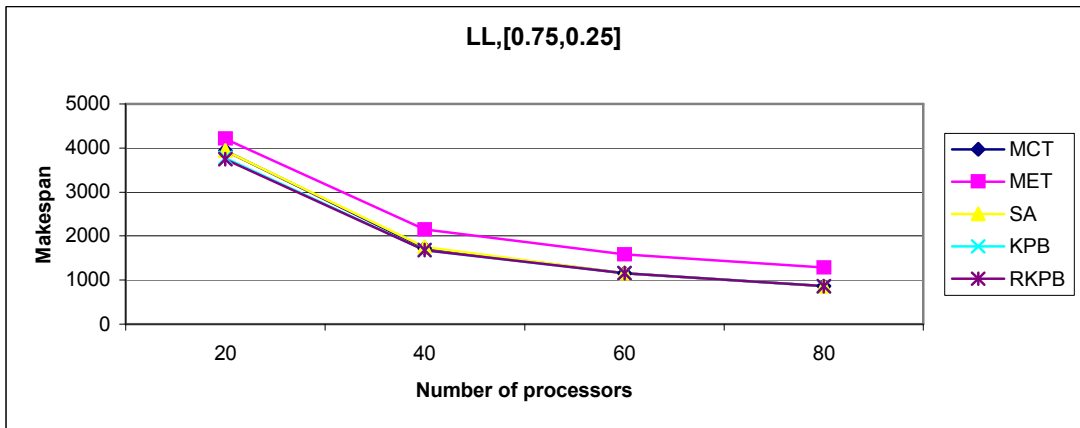


Figure V.43. Makespan vs. Number of processors, LoLo heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

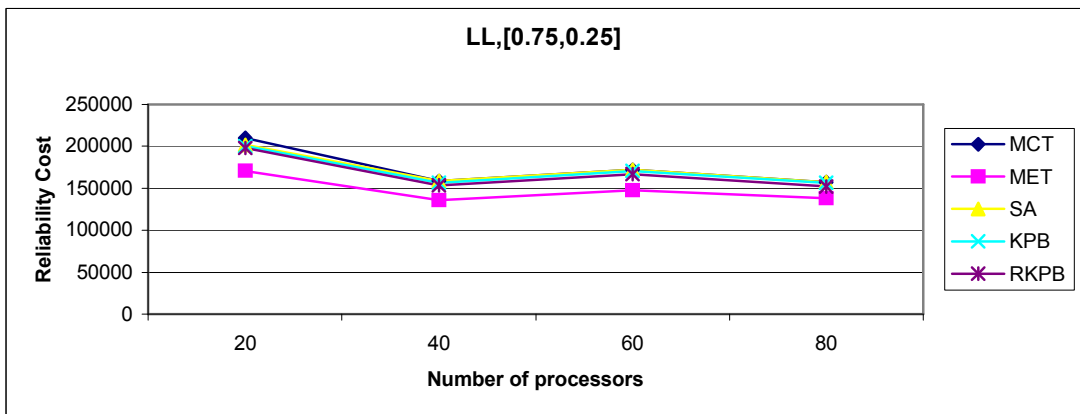


Figure V.44. Reliability Cost vs. Number of processors, LoLo heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

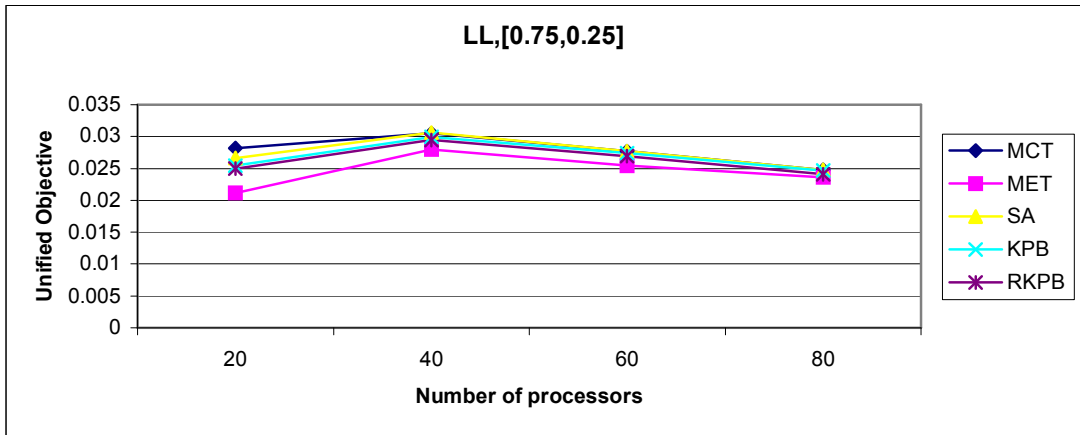


Figure V.45. Unified Objective vs. Number of processors, LoLo heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

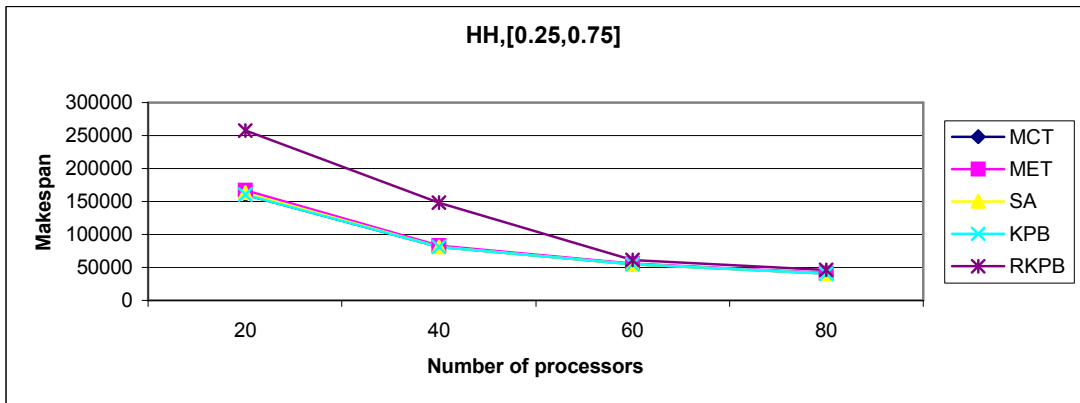


Figure V.46. Makespan vs. Number of processors, HiHi heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

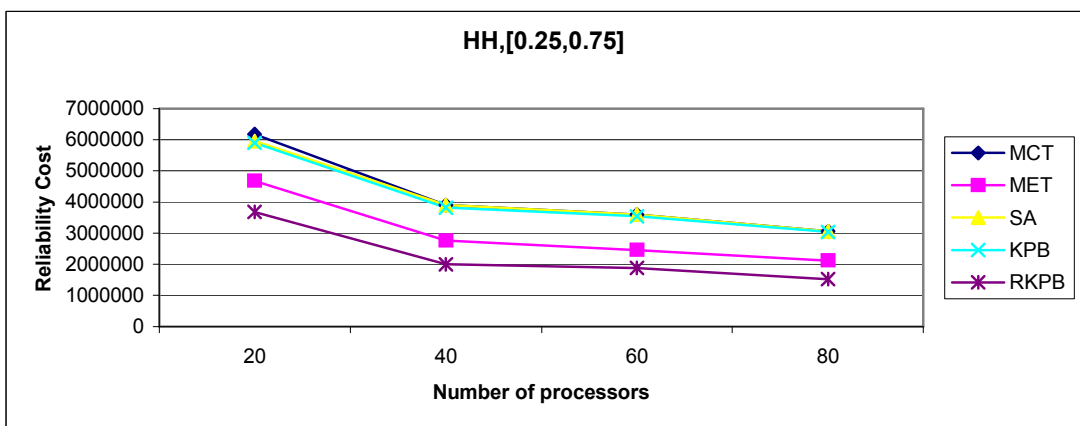


Figure V.47. Reliability Cost vs. Number of processors, HiHi heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

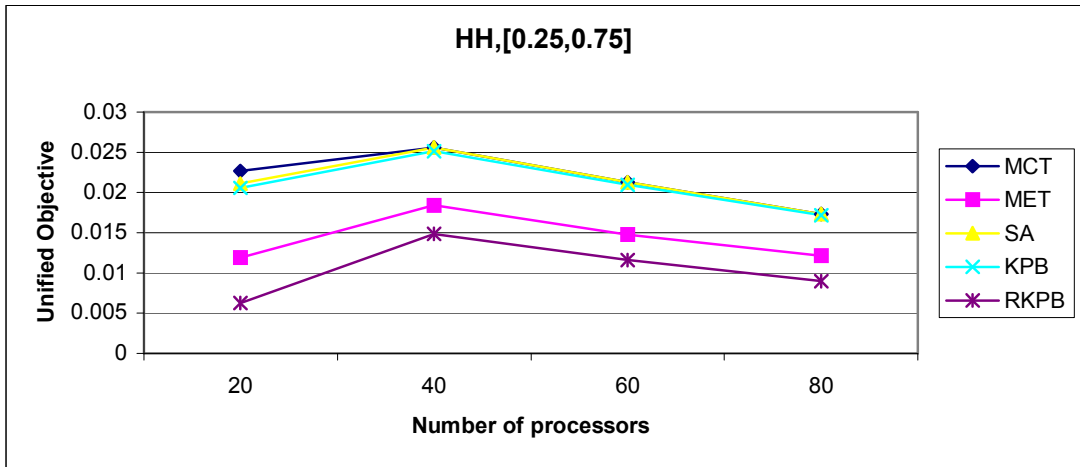


Figure V.48. Unified Objective vs. Number of processors, HiHi heterogeneity,  $w_1=0.25$ ,  $w_2=0.75$

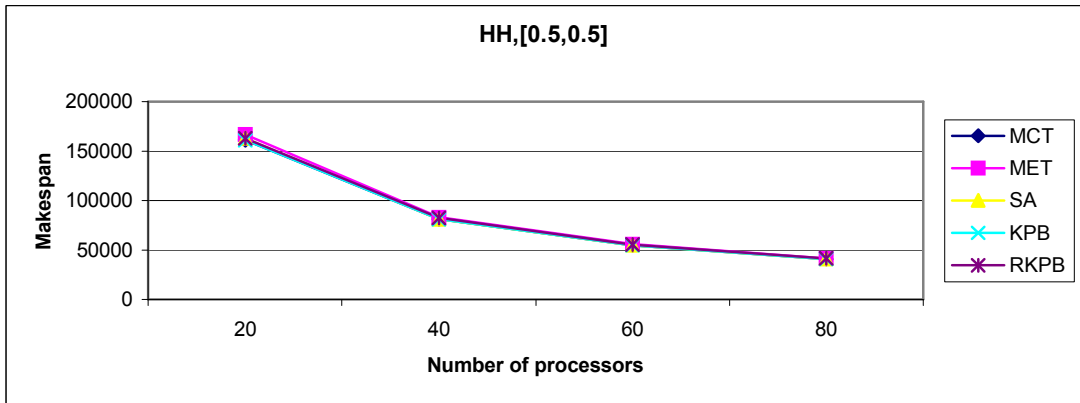


Figure V.49. Makespan vs. Number of processors, HiHi heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

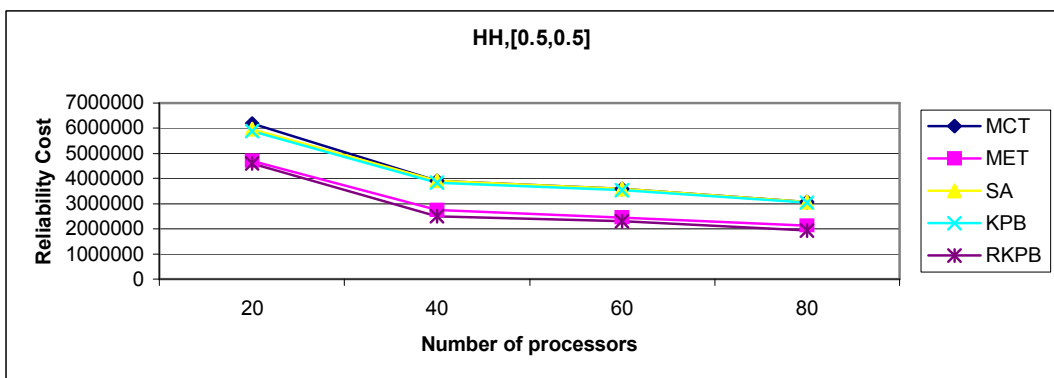


Figure V.50. Reliability Cost vs. Number of processors, HiHi heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

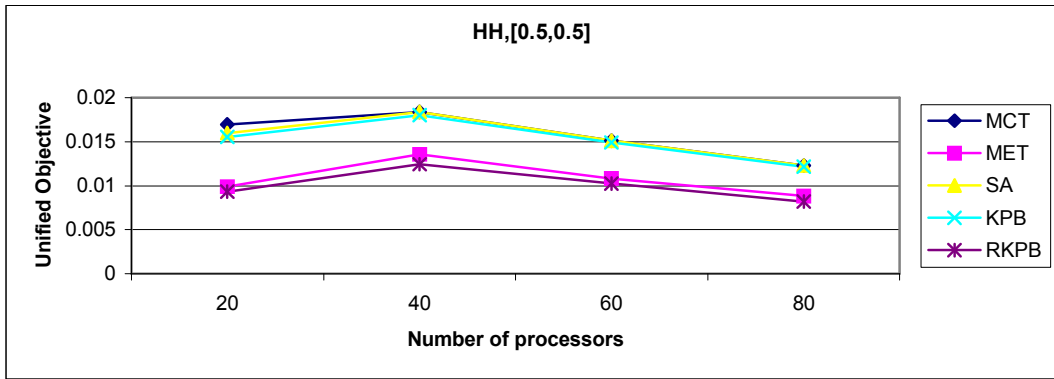


Figure V.51. Unified Objective vs. Number of processors, HiHi heterogeneity,  $w_1=0.5$ ,  $w_2=0.5$

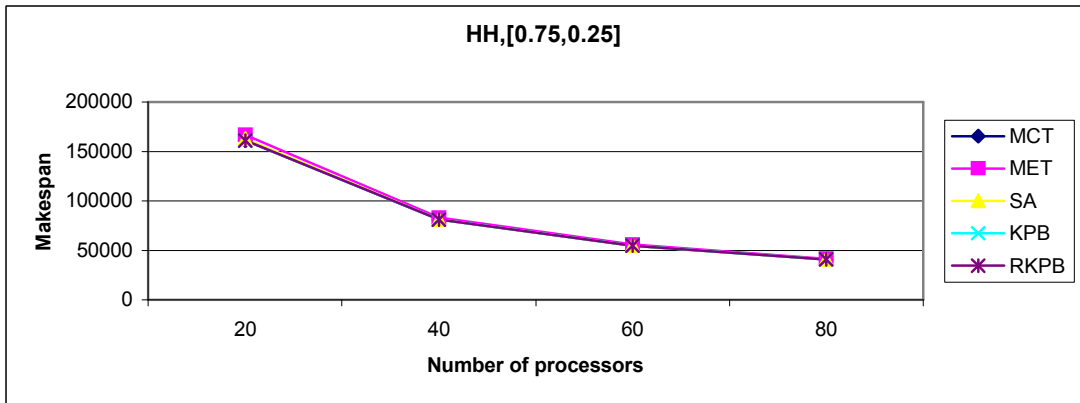


Figure V.52. Makespan vs. Number of processors, HiHi heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

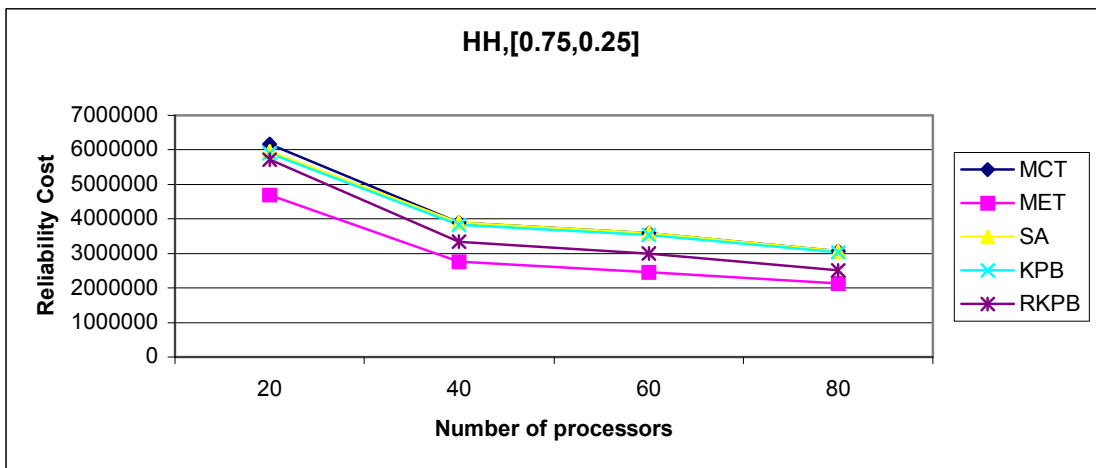


Figure V.53. Reliability Cost vs. Number of processors, HiHi heterogeneity,  $w_1=0.75$ ,  $w_2=0.25$

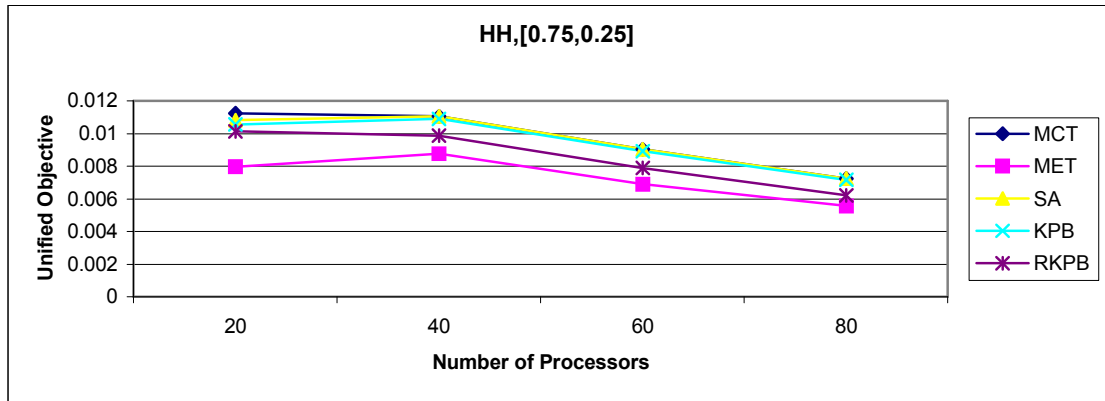


Figure V.54. Unified Objective vs. Number of processors, HiHi heterogeneity,  $w_1=0.75, w_2=0.25$

In the fourth test case the effects of changing weights are observed. Number of tasks differentiate between 100 and 1000,  $\alpha$  between 0.1 and 0.75 and four task-machine heterogeneity cases are applied. Processor number is kept at 40. 25 different task graph is tested for each case. Average of all cases are taken and three performance metric is observed when the weights change. 5 different weights are used:  $W_1, W_2, W_3, W_4, W_5$  ranging in  $\{(0,1), (0.25,0.75), (0.5,0.5), (0.75,0.25), (1,0)\}$ . According to that as weights changes from  $W_1$  to  $W_5$ , makespan of RKPb decreases and becomes equal to KPB at last. The reliability cost of RKPb increases to KPB's again. For  $W_1, W_2$  and  $W_3$  it outperforms all other algorithms. For  $W_4$  and  $W_5$  MET outperforms RKPb. For the unified objective as the weights change RKPb outperforms the other algorithms for  $W_1, W_2, W_3$  and  $W_5$ . In  $W_4$  MET slightly passes RKPb.

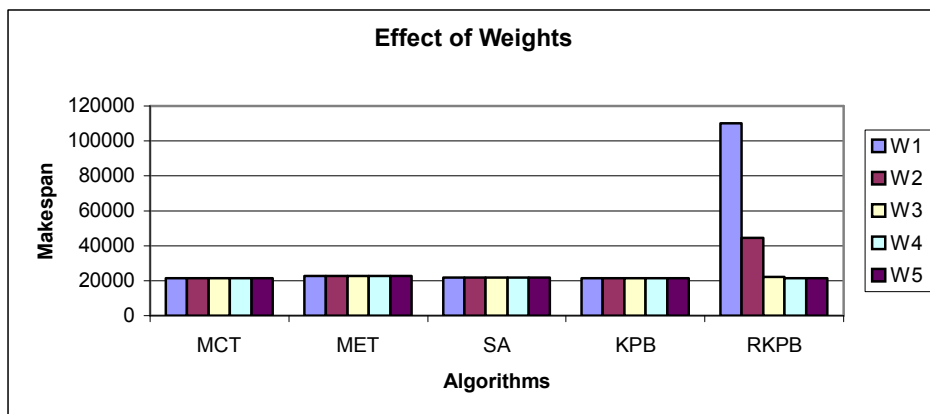


Figure V.55. Effect of weights on makespan

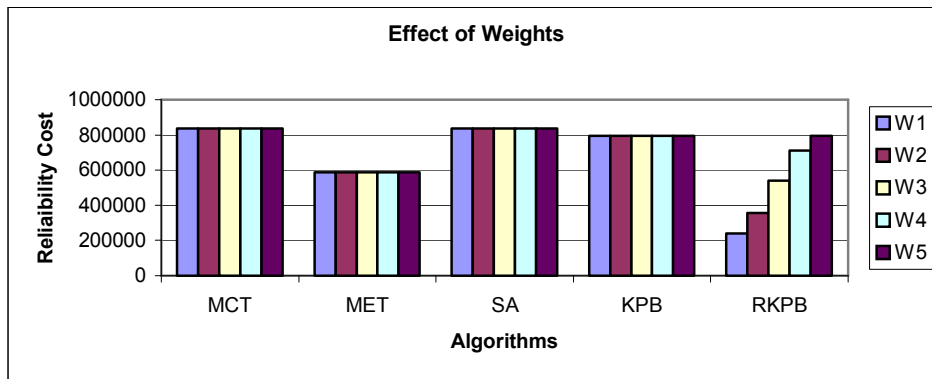


Figure V.56. Effect of weights on Reliability Cost

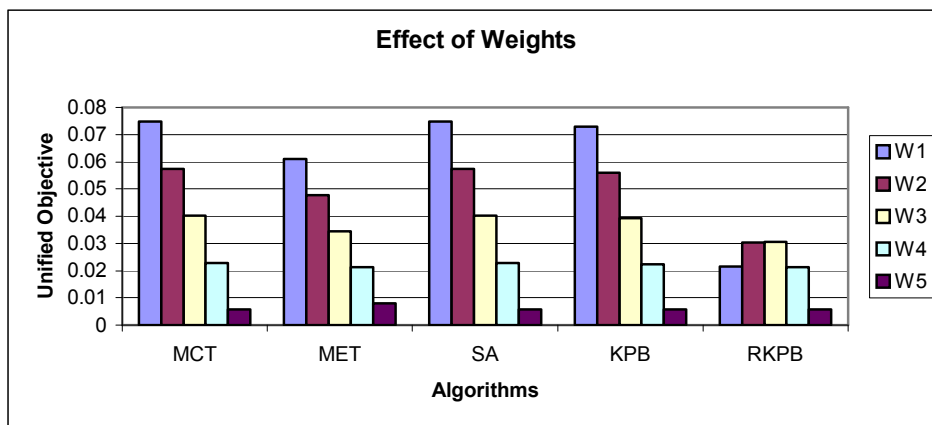


Figure V.57. Effect of weights on Unified Objective

## V.2. PERFORMANCE EVALUATION OF BATCH MODE ALGORITHMS

The parameter settings used in this section is similar to the one that were used in immediate mode algorithms. In addition to the ones used batch size of 10,25,50 and 100 values are tested. For the memetic algorithm different combinations of 2 crossovers and 3 mutations are used on a population of size 200. 1000 iterations are made in each application of memetic algorithm on a batch.

### V.2.1. RESULTS AND DISCUSSION

In this section test case combinations are given and their results are discussed for evaluation. The test cases are the same as the ones used in the immediate mode.

The first test case mainly is aimed to present the effect of number of tasks on the performance of algorithms for different combinations of task machine heterogeneity, batch sizes, unified objective weights and crossover-mutation types. The processor number is kept at 40 and  $\alpha$  at 0.25. Two different heterogeneity cases are applied, LoLo and HiHi. Batch sizes differ in range [10,20,50]. Number of tasks change between 100 and 1000. Three different weight combinations are applied:  $W_1=\{w_1,w_2\}=\{0.25,0.75\}$ ,  $W_2=\{w_1,w_2\}=\{0.5,0.5\}$ ,  $W_3=\{w_1,w_2\}=\{0.75,0.25\}$ .

For LoLo heterogeneity case, when  $w_1$  is applied MA gives the best result for reliability cost and unified objective. However in terms of makespan it gives the worst result. The other algorithms gives similar results except that when the number of tasks is high Sufferage algorithm perform the worst. When  $W_2$  is applied, MA gives close results in terms of makespan and better results in terms of reliability cost and unified objective. The unified objective value is this time closer to the other algorithms. In case of  $W_3$  applied, MA gives better results in terms of makespan when the number of tasks are small and better than Sufferage when task number is high. For reliability cost it still gives better results and but the difference is small. For the unified objective MA is still better when the number of tasks are small. For the HiHi heterogeneity case the results are similar.

The best performances of MA is gained when uniform crossover and mutation type 1 is applied. Again one point order crossover and mutation type 1 follows it.

It is also observed that as batch size increases there is a significant decrease in makespan, reliability cost and unified objective.

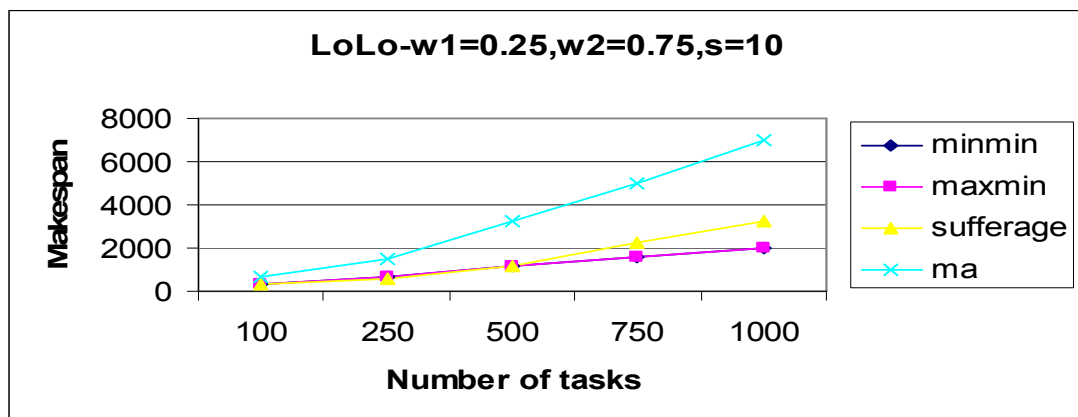


Figure V.2.1. Makespan vs. Number of tasks for batch size of 10,  $W_1$ , LoLo

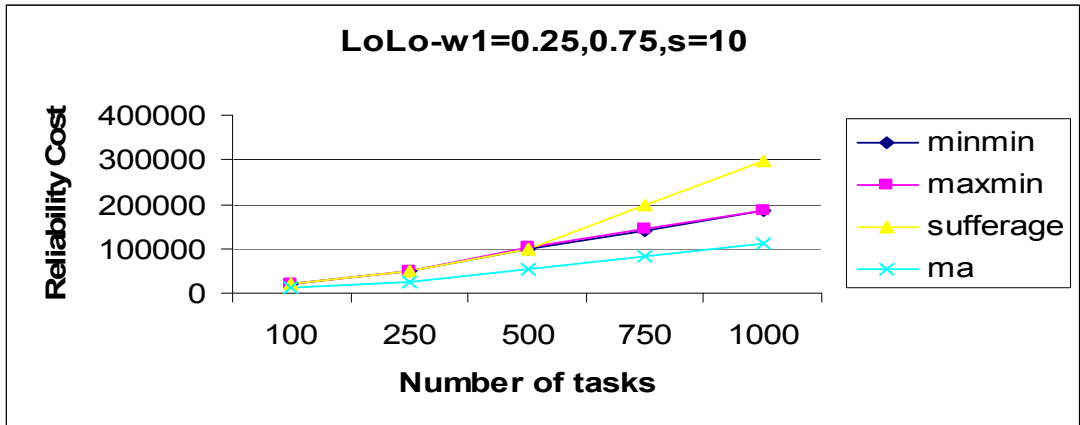


Figure V.2.2. Reliability cost vs. Number of tasks for batch size of 10,  $W_1$ , LoLo

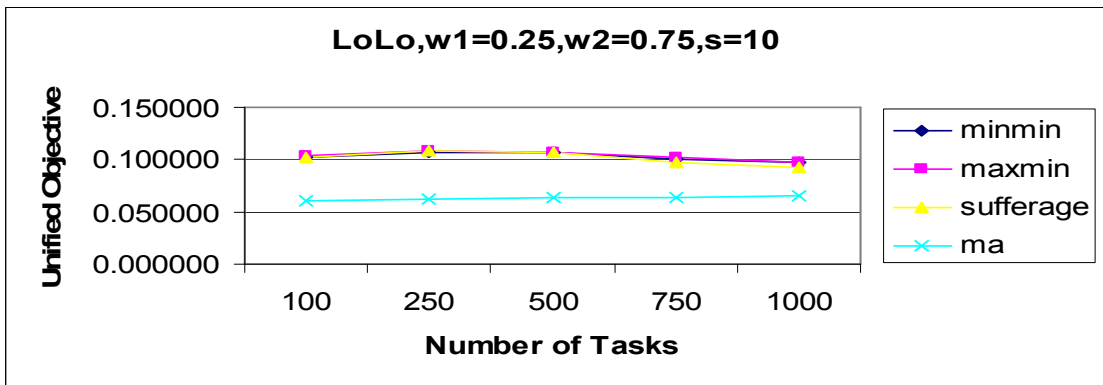


Figure V.2.3. Unified Objective vs. Number of tasks for batch size of 10,  $W_1$ , LoLo

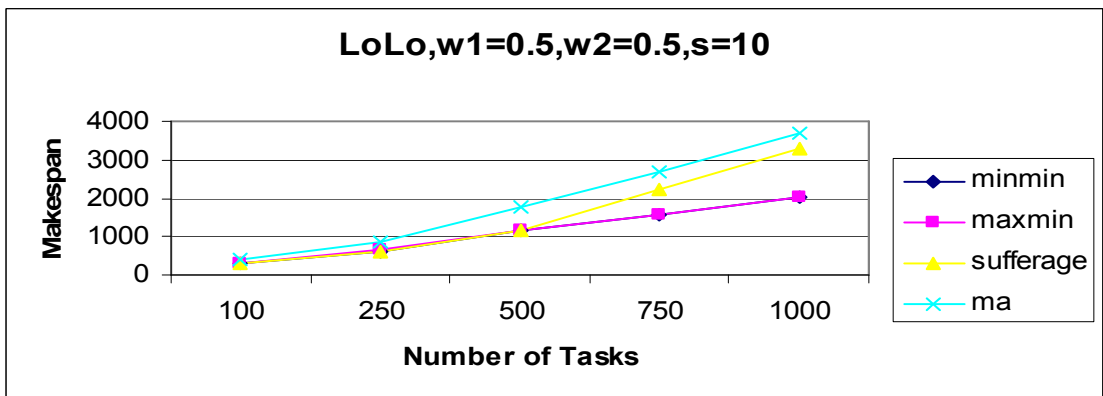


Figure V.2.4. Makespan vs. Number of tasks for batch size of 10,  $W_2$ , LoLo

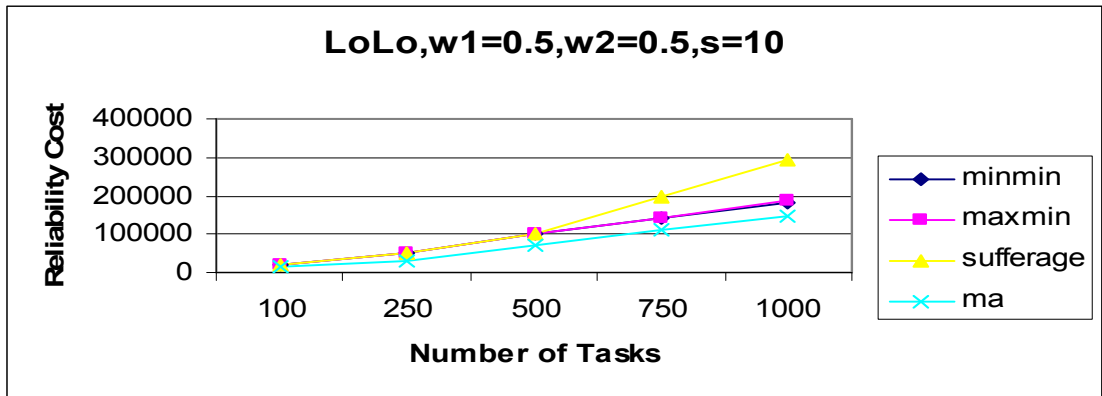


Figure V.2.5. Reliability cost vs. Number of tasks for batch size of 10,  $W_2$ , LoLo

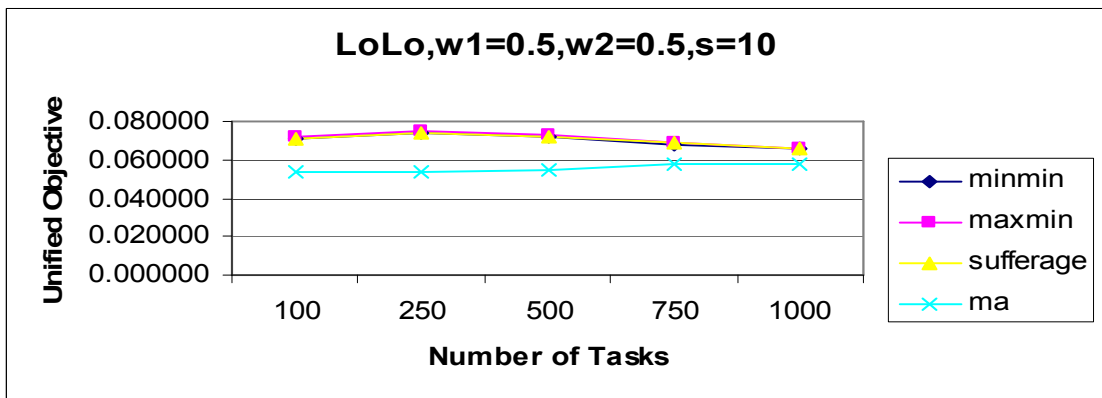


Figure V.2.6. Unified Objective vs. Number of tasks for batch size of 10,  $W_2$ , LoLo

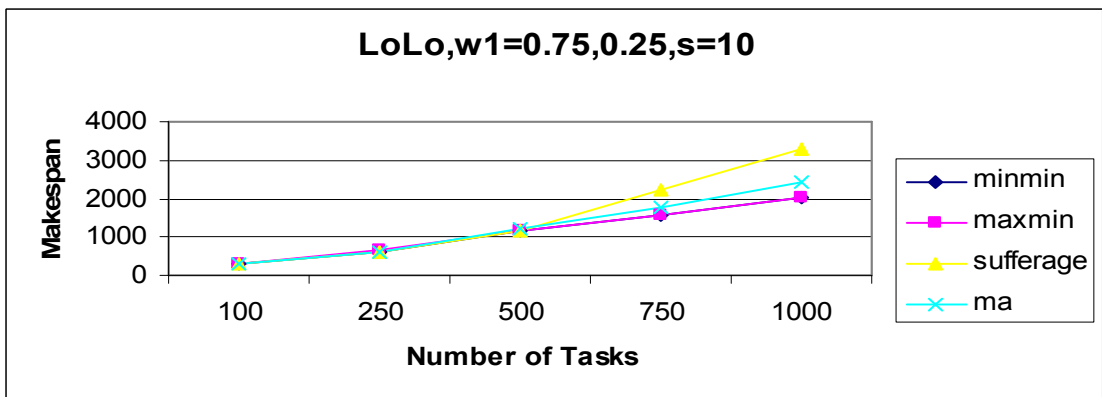


Figure V.2.7. Makespan vs. Number of tasks for batch size of 10,  $W_3$ , LoLo

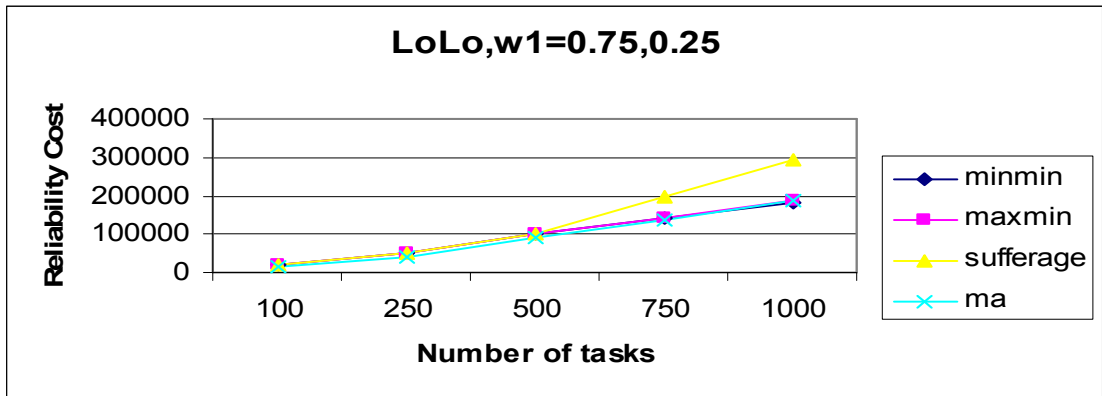


Figure V.2.8. Reliability cost vs. Number of tasks for batch size of 10,  $W_3$ , LoLo

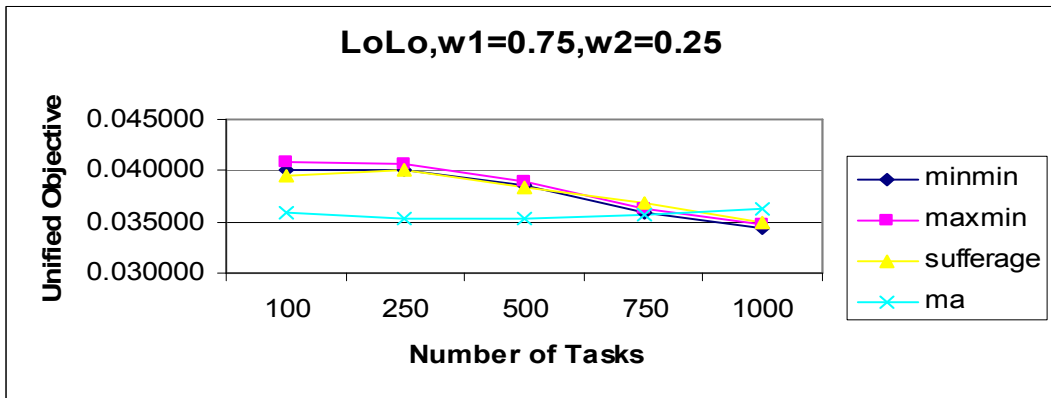


Figure V.2.9. Unified Objective vs. Number of tasks for batch size of 10,  $W_3$ , LoLo

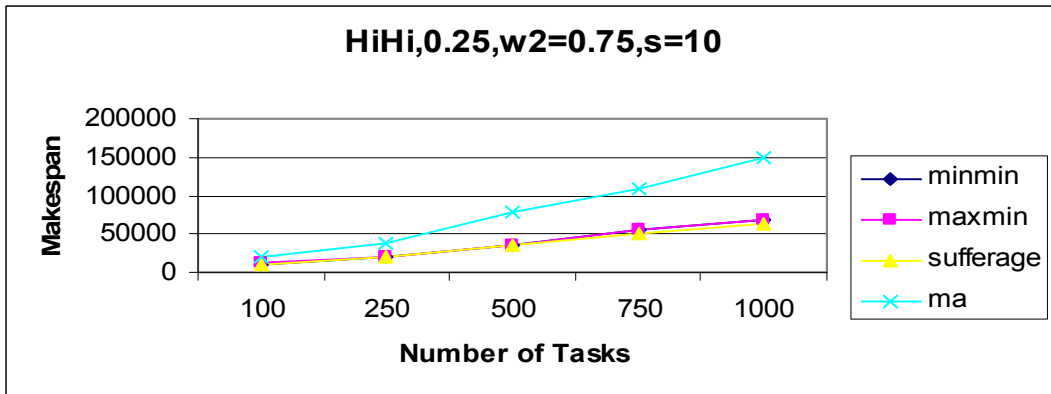


Figure V.2.10. Makespan vs. Number of tasks for batch size of 10,  $W_1$ , HiHi

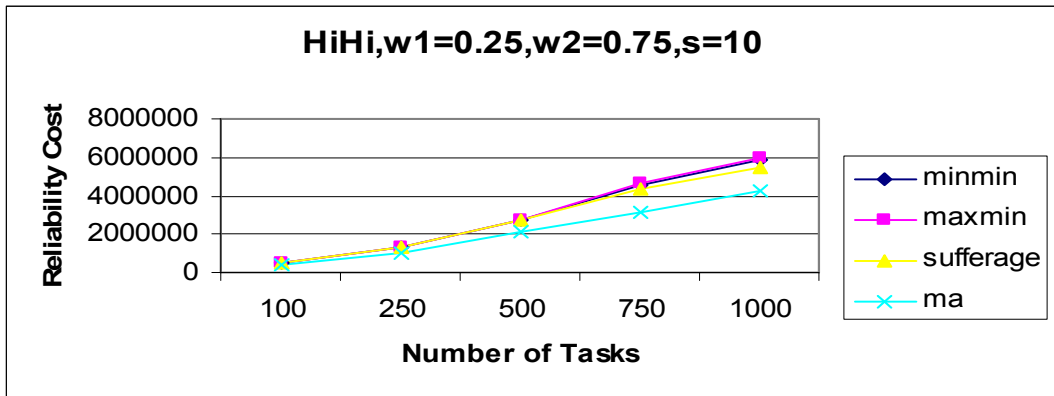


Figure V.2.11. Reliability cost vs. Number of tasks for batch size of 10,  $W_1$ , HiHi

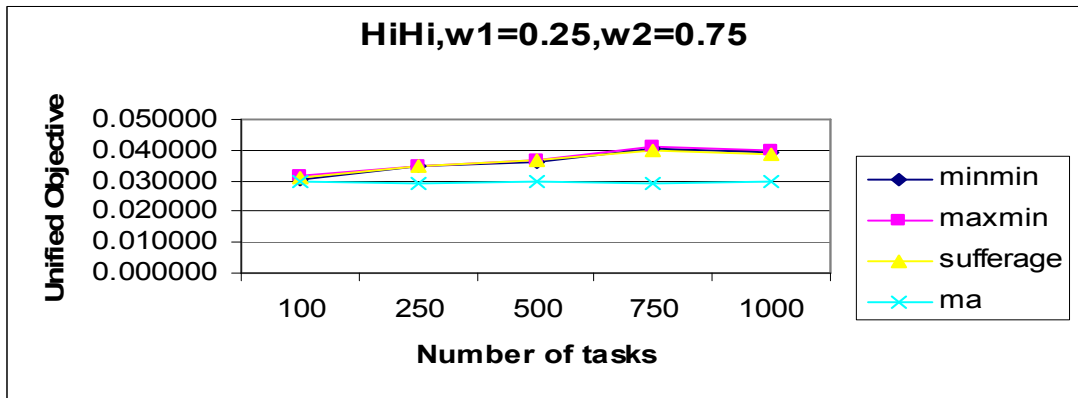


Figure V.2.12. Unified Objective vs. Number of tasks for batch size of 10,  $W_1$ , HiHi

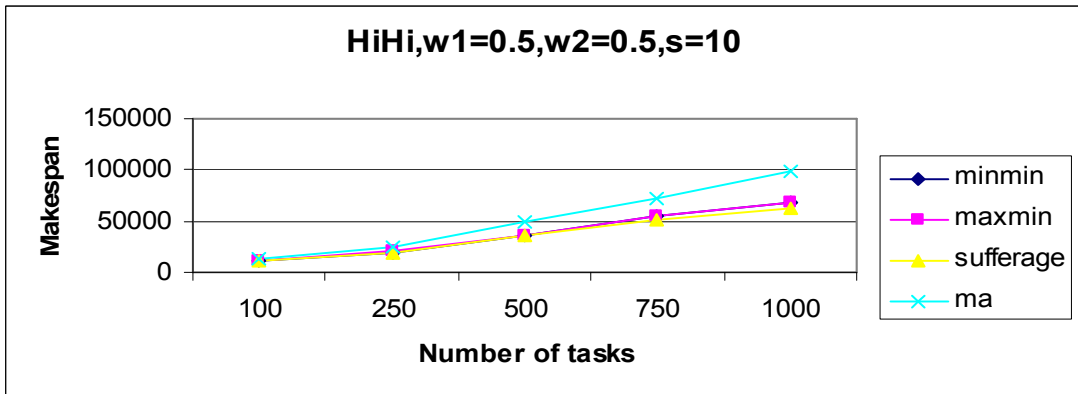


Figure V.2.13. Makespan vs. Number of tasks for batch size of 10,  $W_2$ , HiHi

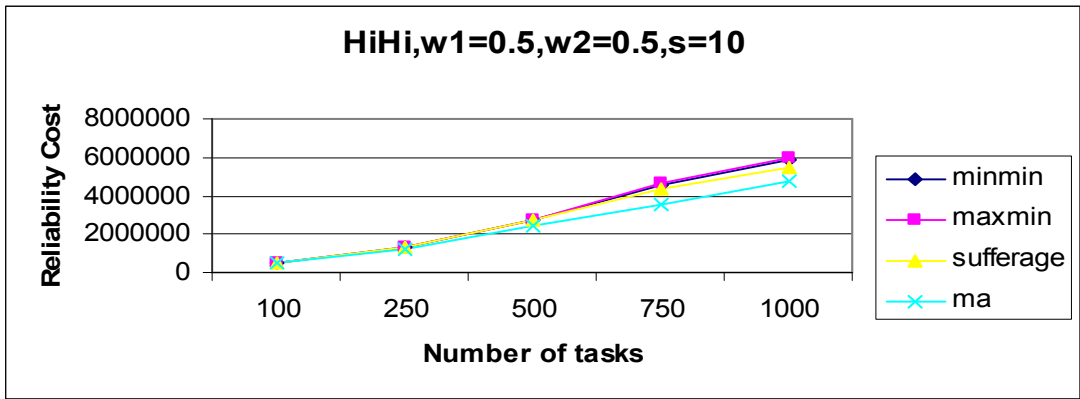


Figure V.2.14. Reliability Cost vs. Number of tasks for batch size of 10,  $W_2$ , HiHi

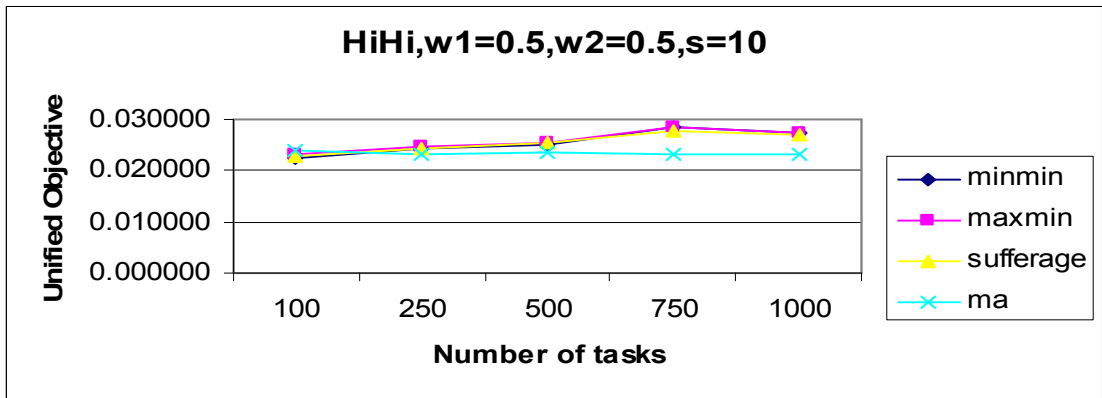


Figure V.2.15. Unified Objective vs. Number of tasks for batch size of 10,  $W_2$ , HiHi

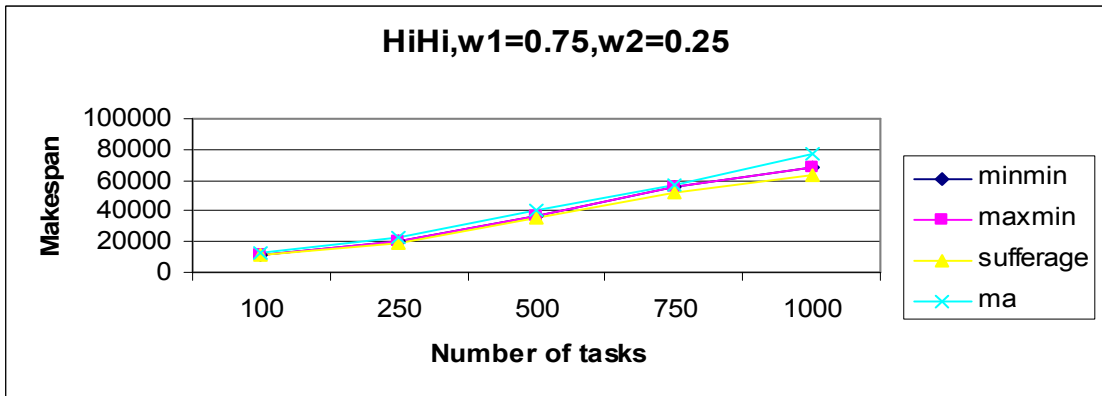


Figure V.2.16. Makespan vs. Number of tasks for batch size of 10,  $W_3$ , HiHi

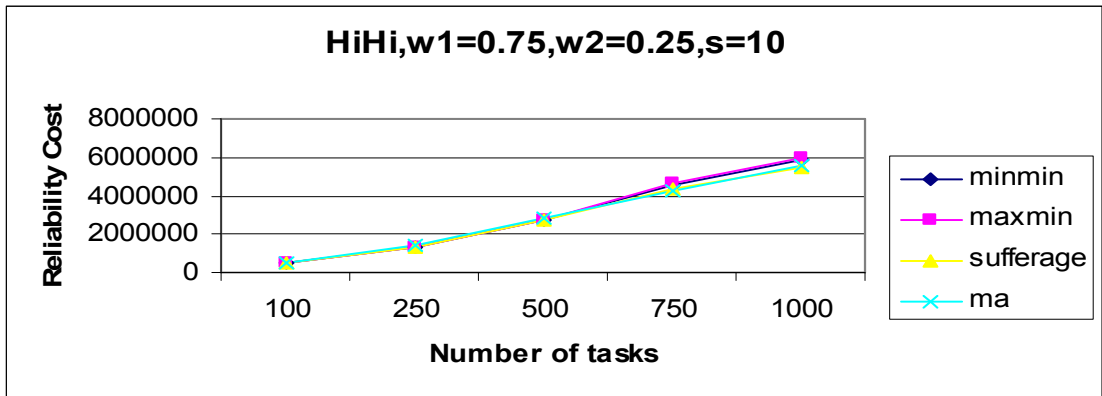


Figure V.2.17. Reliability cost vs. Number of tasks for batch size of 10,  $W_3$ , HiHi

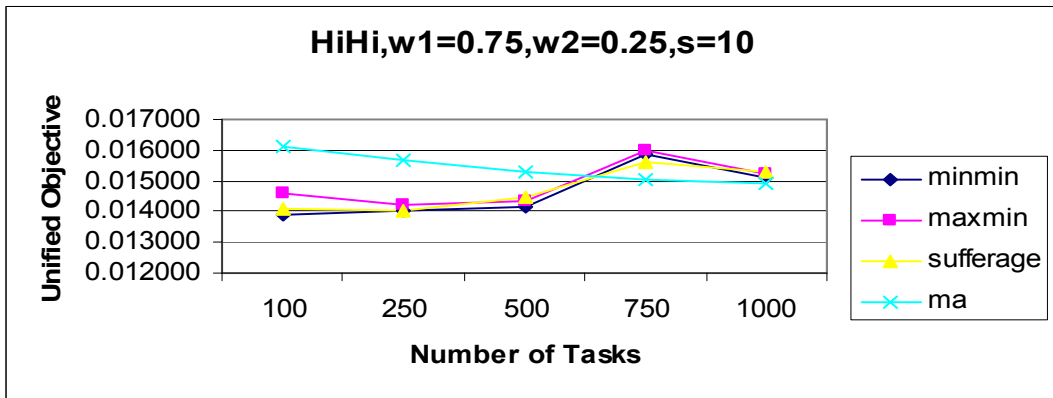


Figure V.2.18. Unified Objective vs. Number of tasks for batch size of 10,  $W_3$ , HiHi

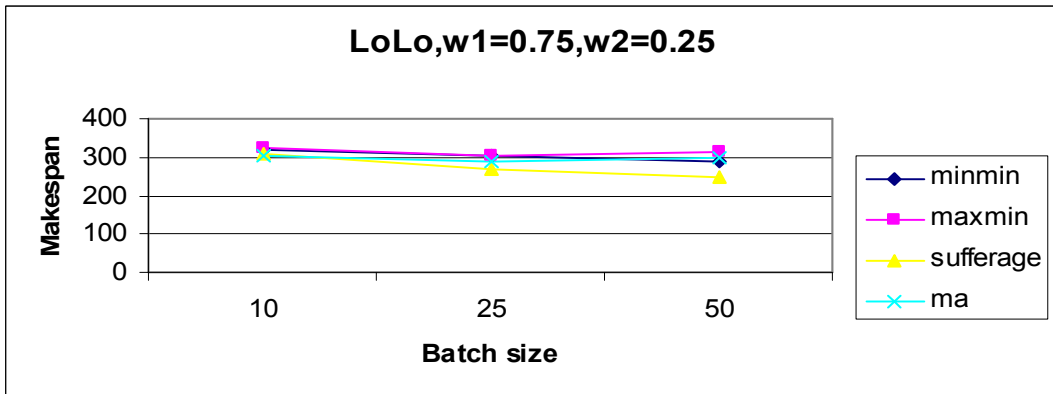


Figure V.2.19. Makespan vs. Batch size for  $W_3$ , HiHi

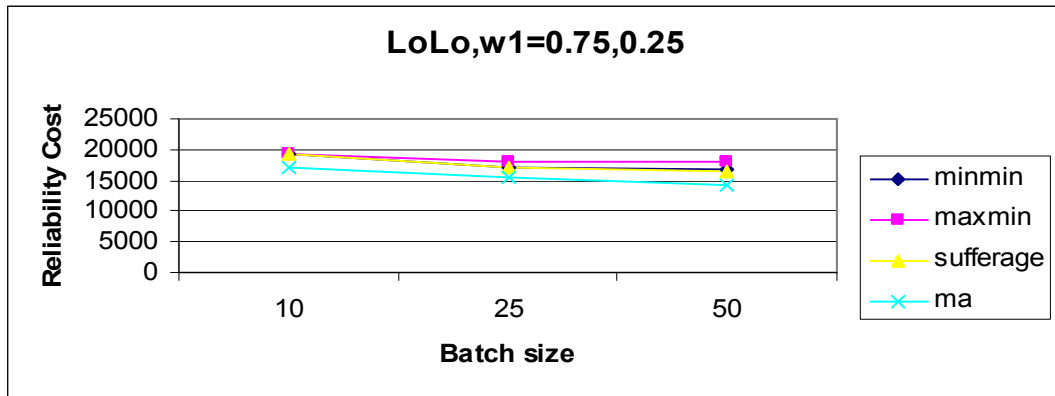


Figure V.2.20. Reliability cost vs. Batch size for  $W_3$ , HiHi

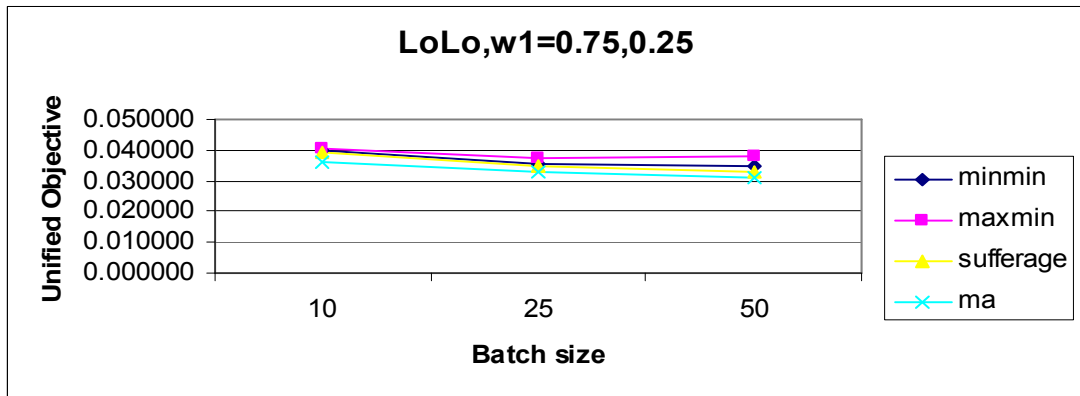


Figure V.2.21. Unified Objective vs. Batch size for  $W_3$ , HiHi

The second test case includes the measurement of the effect of processor number on the makespan, reliability cost and unified objective metrics. The number of tasks is held constant at 1000. The processor number changes in the range [20,40,60,80]. Batch sizes change in the range [25,50,100]. A total of 25 different task set is used for each parameter setting. The weight ratios are the same as in the first test case. LoLo and HiHi task machine heterogeneity cases are considered.

In LoLo task machine heterogeneity case, when  $W_1$  is applied, the makespan of MA is worse than all the other algorithms but gives the best result in terms of reliability cost and unified objective. The curve for the makespan tends to decrease as the number of processors increases. That is the result of more parallelism. But the curve for reliability cost does not show a definite characteristics because the number of processors does not effect reliability cost computation. The only effective factor here is the heterogeneity of processors on tasks. When  $W_2$  is applied, MA gives a closer result

to the other results in terms of makespan, again gives the best results for reliability cost and unified objective. For  $W_3$ , MA gives competitive results for makespan but its reliability cost performance decreases so the unified objective at some points can give slightly worse results. For HiHi heterogeneity case the performance of MA significantly decreases in terms of reliability cost when the number of processors is high in all weight ratio combinations.

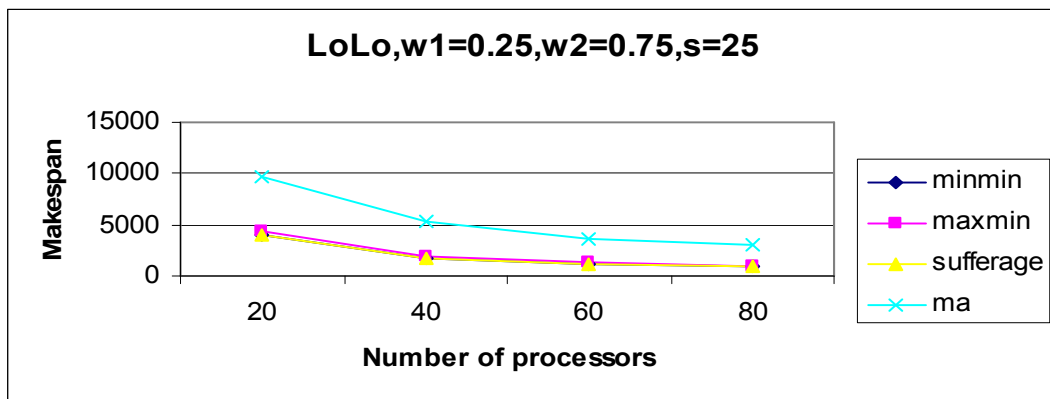


Figure V.2.22. Makespan vs. Number of processors for batch size=25,  $W_1$ , LoLo

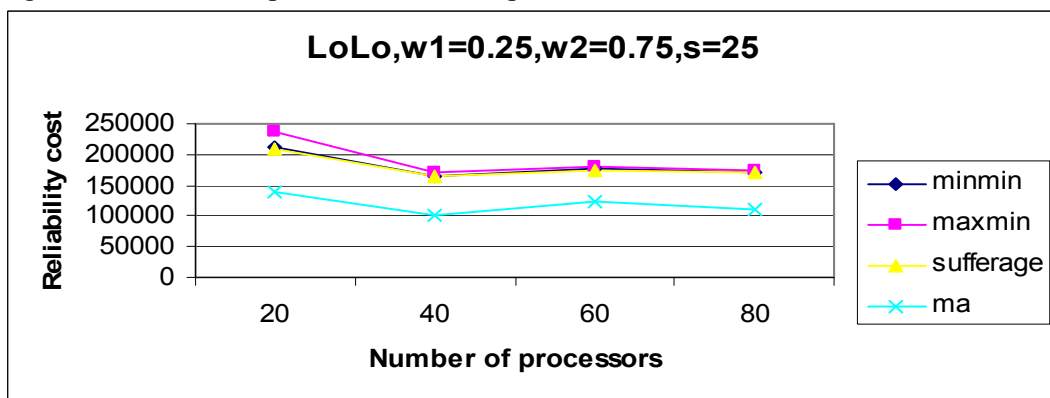


Figure V.2.23. Reliability Cost vs. Number of processors for batch size=25,  $W_1$ , LoLo

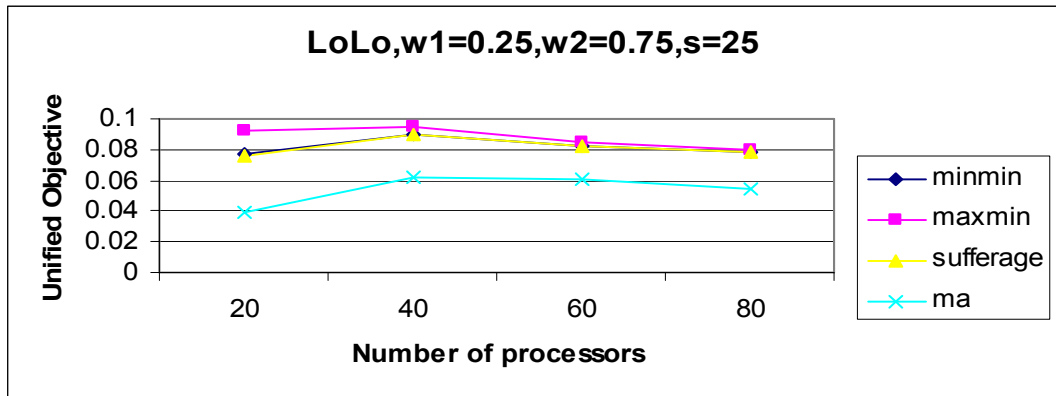


Figure V.2.24. Unified Objective vs. Number of processors for batch size=25,  $W_1$ , LoLo

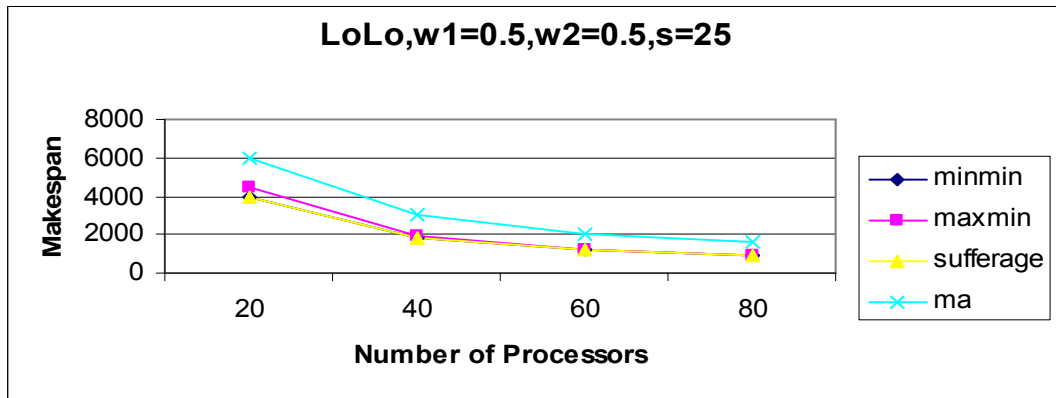


Figure V.2.25. Makespan vs. Number of processors for batch size=25,  $W_2$ , LoLo

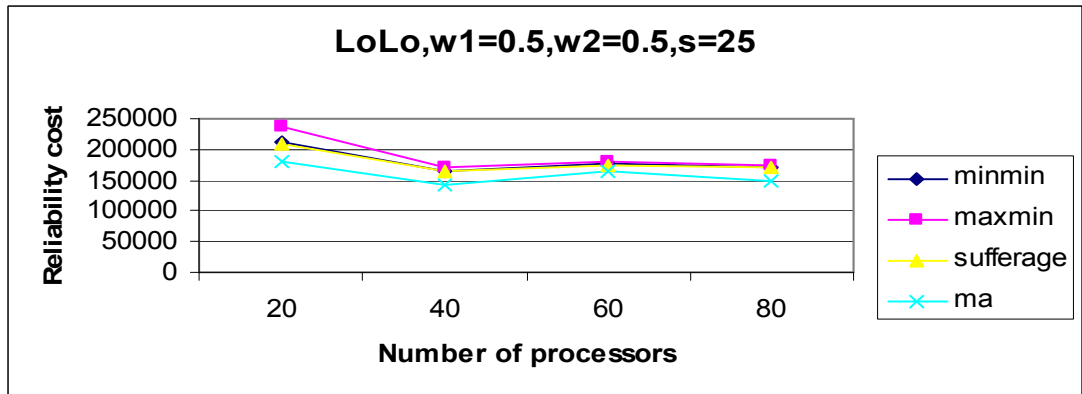


Figure V.2.26. Reliability cost vs. Number of processors for batch size=25,  $W_2$ , LoLo

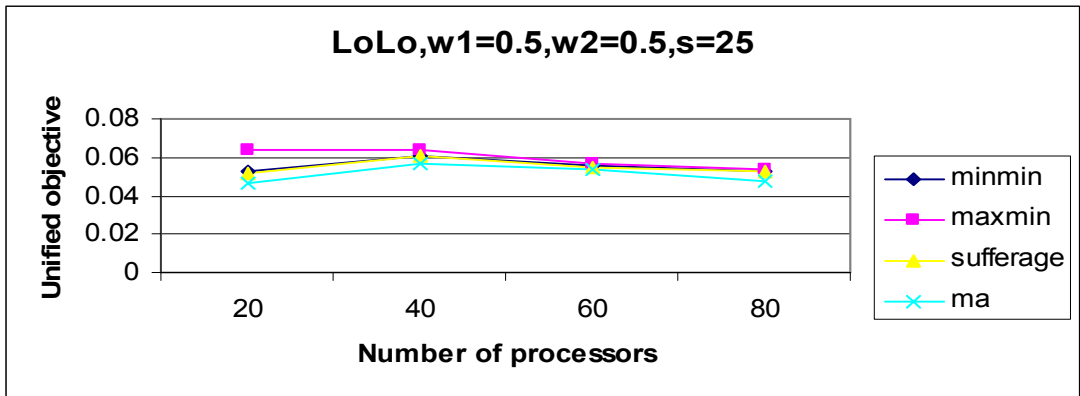


Figure V.2.27. Unified Objective vs. Number of processors for batch size=25,  $W_2$ , LoLo

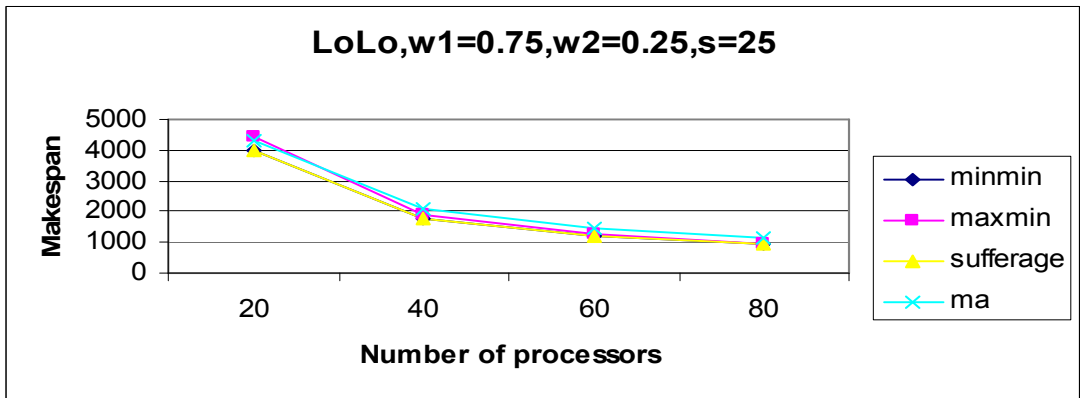


Figure V.2.28. Makespan vs. Number of processors for batch size=25,  $W_3$ , LoLo

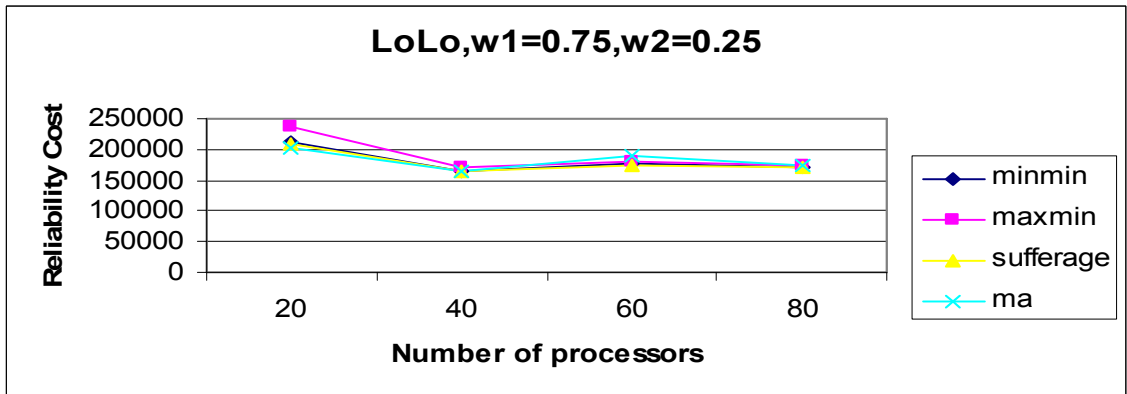


Figure V.2.29. Reliability cost vs. Number of processors for batch size=25,  $W_3$ , LoLo

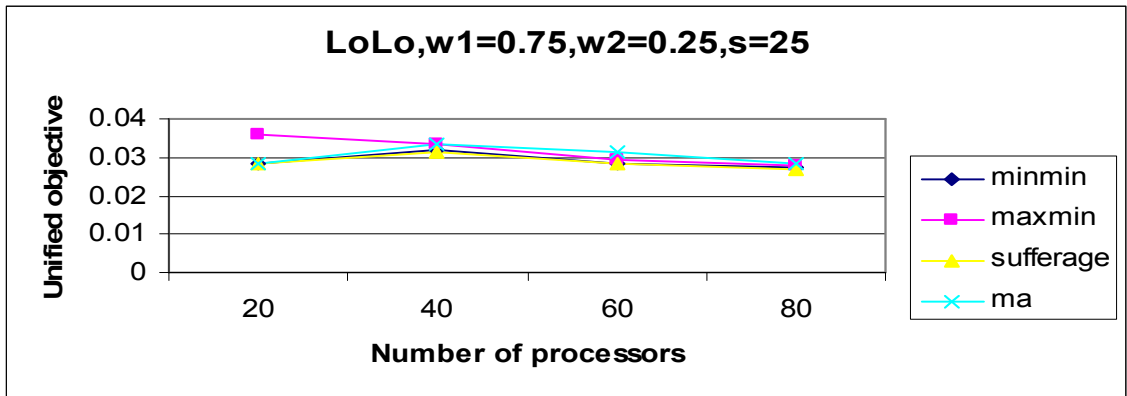


Figure V.2.30. Unified Objective vs. Number of processors for batch size=25,  $W_3$ , LoLo

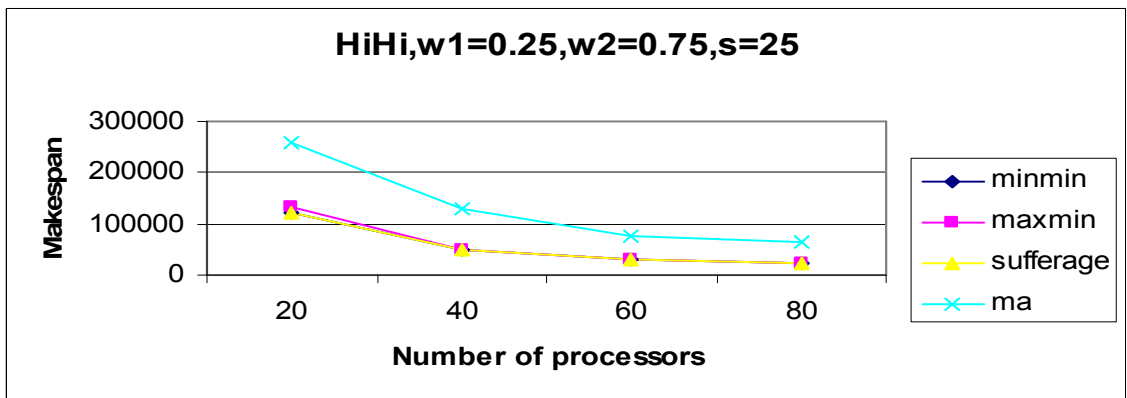


Figure V.2.31. Makespan vs. Number of processors for batch size=25,  $W_1$ , HiHi

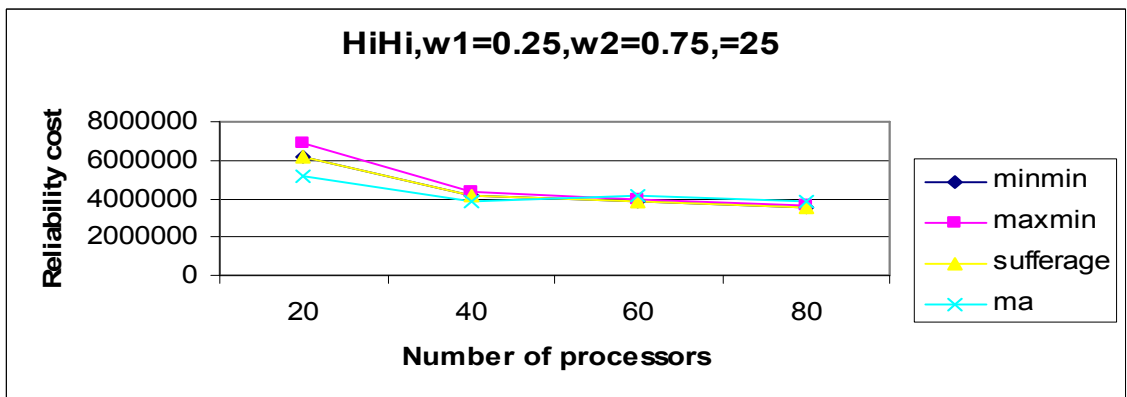


Figure V.2.32. Reliability cost vs. Number of processors for batch size=25,  $W_1$ , HiHi

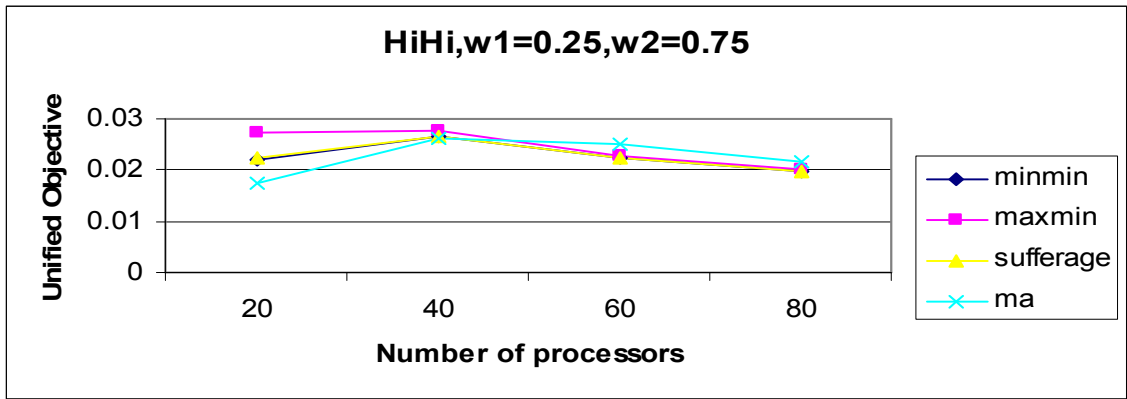


Figure V.2.33. Unified Objective vs. Number of processors for batch size=25,  $W_1$ , HiHi

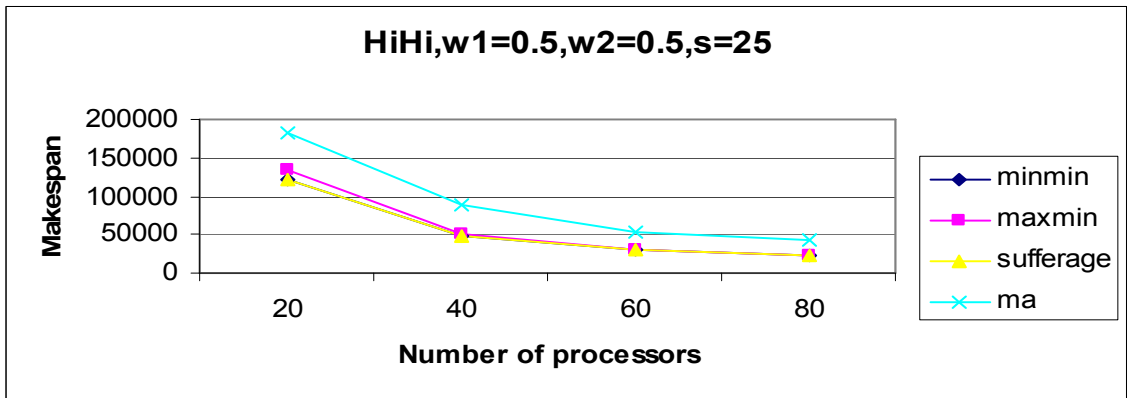


Figure V.2.34. Makespan vs. Number of processors for batch size=25,  $W_2$ , HiHi

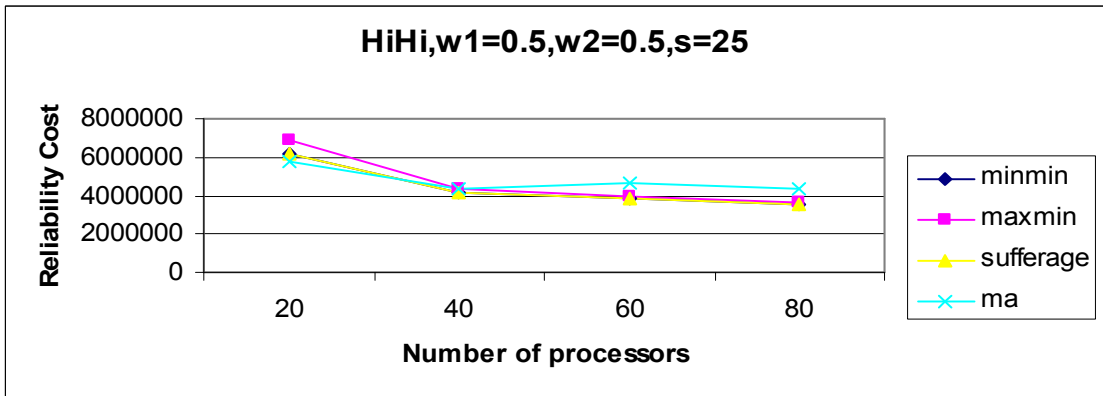


Figure V.2.35. Reliability cost vs. Number of processors for batch size=25,  $W_2$ , HiHi

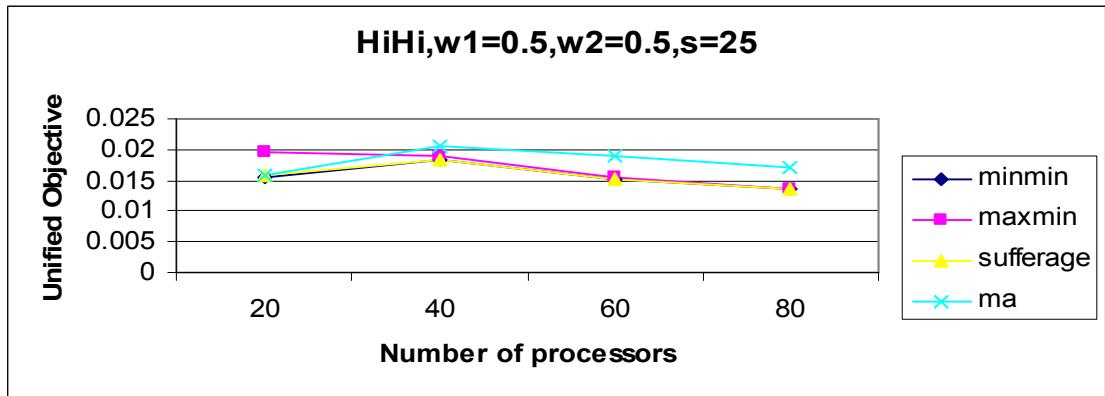


Figure V.2.36. Unified Objective vs. Number of processors for batch size=25,  $W_2$ , HiHi

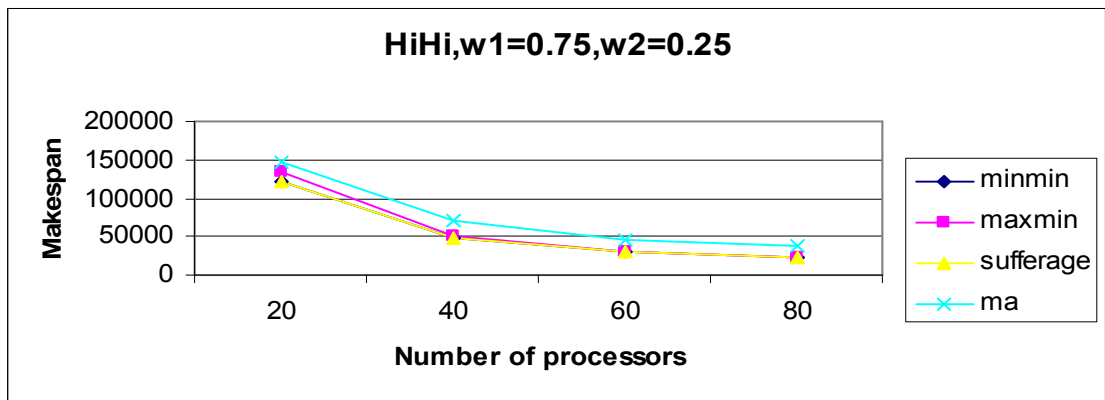


Figure V.2.37. Makespan vs. Number of processors for batch size=25,  $W_3$ , HiHi

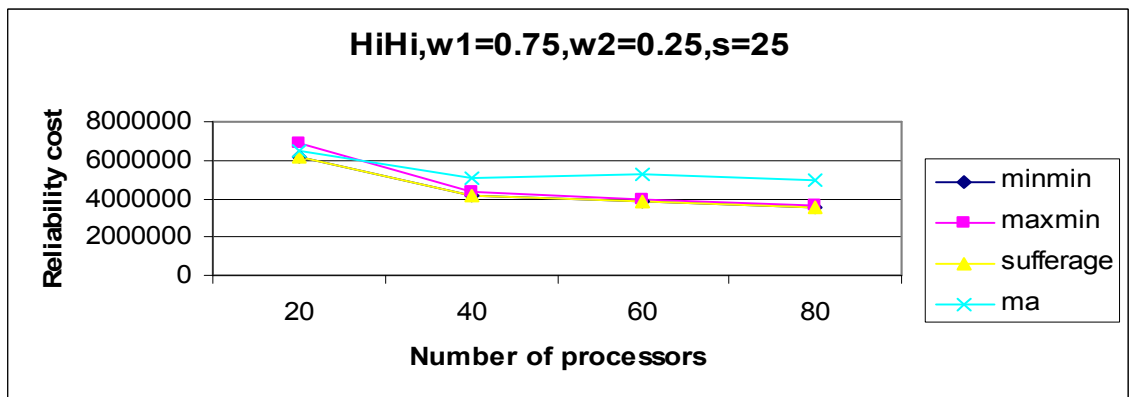


Figure V.2.38. Reliability cost vs. Number of processors for batch size=25,  $W_3$ , HiHi

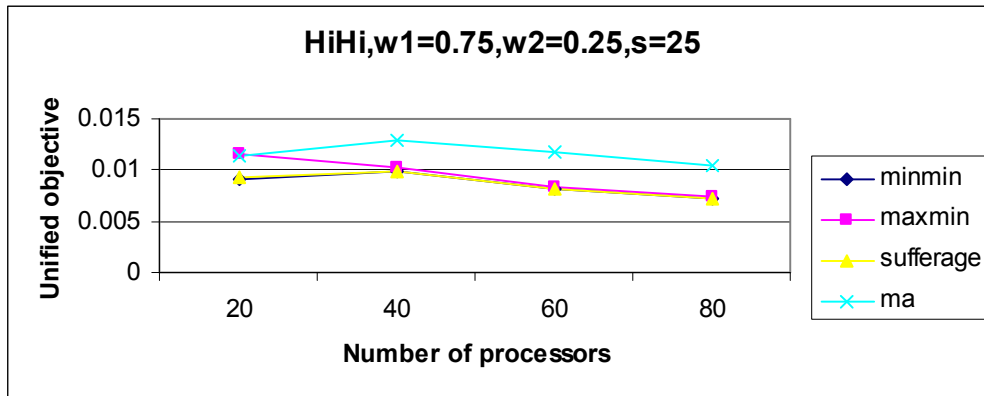


Figure V.2.39. Unified objective vs. Number of processors for batch size=25,  $W_3$ , HiHi

The third test case includes the effects of weights on makespan, reliability cost and unified objective. Five weight combinations are used,  $W_1=\{0,1\}$ ,  $W_2=\{0.25,0.75\}$ ,  $W_3=\{0.5,0.5\}$ ,  $W_4=\{0.75,0.25\}$ ,  $W_5=\{1,0\}$ . Number of tasks, heterogeneity and batch size, crossover and mutation types has changed in a range similar to the previous test cases and an average result is calculated. Total of 25 different task sets are used.

MA gives very bad results for makespan, when  $W_1$  is the case. But it is reduced significantly when the weight ranges from  $W_1$  to  $W_4$ . For reliability cost it gives bet result for  $W_1, W_2$  and  $W_3$ . In other cases it gives higher results with respect to other algorithms. For the unified objective it gives the best results for  $W_1, W_2, W_3$  and  $W_4$ . Only in  $W_5$  it gives bad result. In conclusion, MA gives good results in most of the parameter cases.

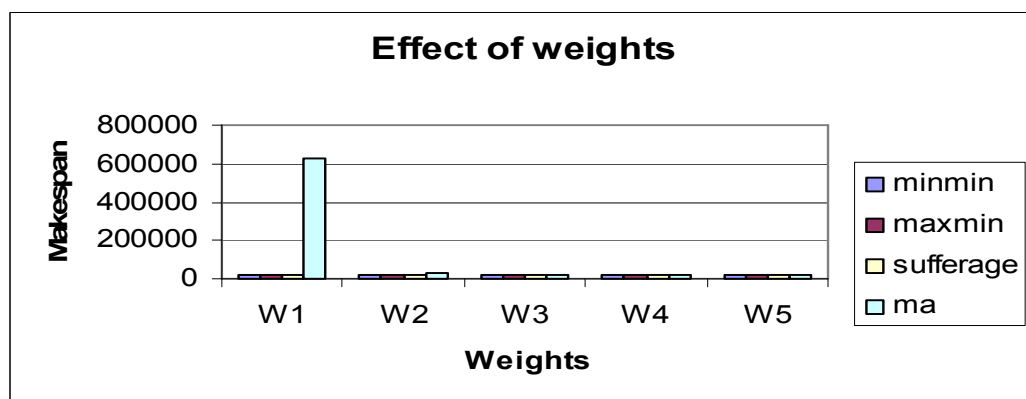


Figure V.2.40. Makespan vs. Weights

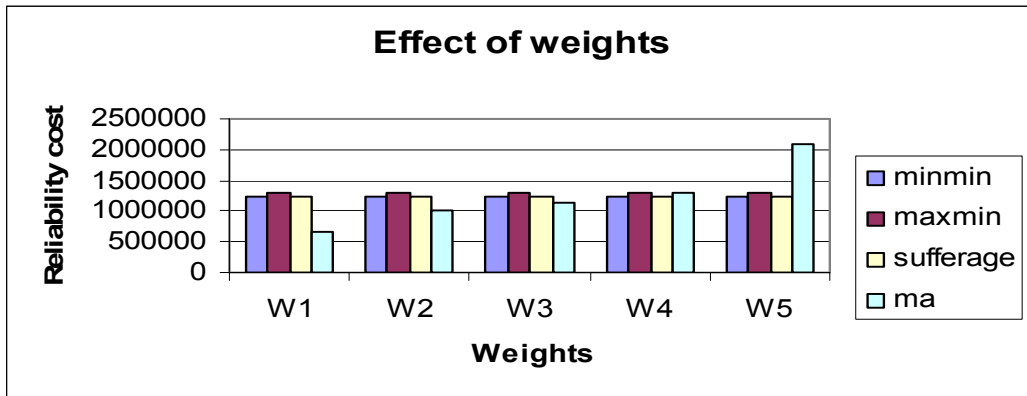


Figure V.2.41. Reliability cost vs. Weights

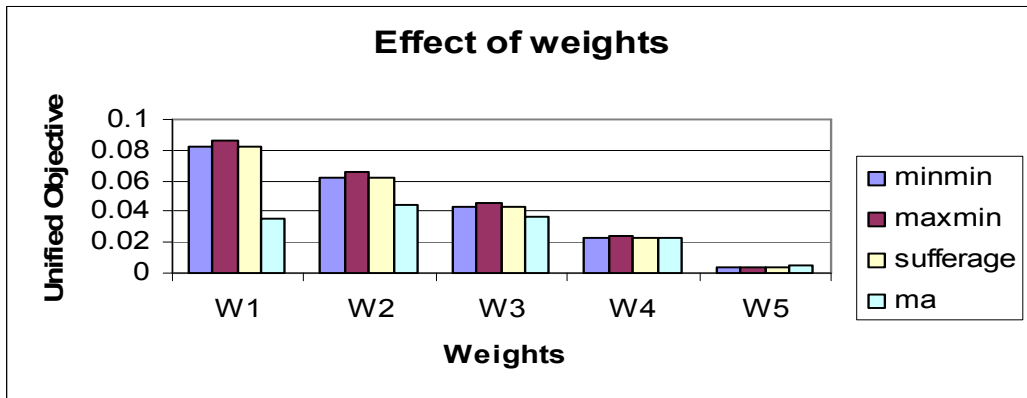


Figure V.2.42. Unified Objective vs. Weights

## **PART VI**

### **CONCLUSIONS**

The dynamic scheduling of independent tasks problem is solved in the literature with only one objective of minimizing makespan, however the possibility of failure of processors was not considered. To be able to provide reliability of application tasks two objectives, makespan and reliability cost should be minimized at the same time.

Throughout the thesis, a weighted formula to minimize two objectives was devised. A reliability computation formula is given for dynamic independent task scheduling problem. Two new algorithms, one immediate and one batch mode memetic algorithm were developed applying this unified objective. The proposed algorithms were compared to the leading 7 dynamic scheduling algorithms.

The experimental results showed that our algorithms give good results in terms of reliability cost and unified objective. The makespan results were close to the other algorithms for equal weight values.

The application of unified objective to the dynamic scheduling of DAG structured task graphs problem stands for a futurework to be completed.

## REFERENCES

- [1] Beaumont, O. ; Boudet, U. ; Robert, Y. : “A Realistic Model and an Efficient Heuristic for Scheduling with Heterogeneous Processors ”, *IPDPS (2002)*.
- [2] Topcuoglu, H. ; Hariri, S. : “Performance-Effective and Low Complexity Task Scheduling for Heterogeneous Computing”, *IEEE Trans. Parallel and Distributed Systems (2002)*, Vol.13, No.3.
- [3] Maheswaran, M. ; Ali, S. ; Siegel, H.J.; Hensgen, D.; Freund, R.F.: “Dynamic Matching and Scheduling of a Class of Independent Tasks Onto Heterogeneous Computing Systems” *Journal of Parallel and Distributed Computing (1999)*, 59(2):107-131.
- [4] Kim, J. ; Shiple, S. ; Siegel, H.J. ; Maciejewski,A.; Braun, T. ;Schneider, M.;Tideman,S.; Chitta,R.; Dilmaghani,R.; Joshi,R.;Kaul,A.; Sharma,A. ; Sripada,S.; Vangari, P.;Yellampalli,S. : “Dynamic Mapping in a Heterogeneous Environment with Tasks Having Priorities and Multiple Deadlines”, *IPDPS (2003)* .
- [5] Jinqun, Z. ; Lina, N. ; Changjun, J. : “A heuristic Scheduling Strategy for Independent Tasks on Grid”, *IEEE HPCASIA (2005)*.
- [6] Maheswaran, M. ; Siegel, H.J. : “A dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems”. *IPPS/SPDP Workshop on Heterogeneous Computing (1998)*.
- [7] Kechadi, M. ; Savvas, I.K. : “Dynamic Task Scheduling for Irregular Network Topologies”, *Elsevier Parallel Computing (2005)*.
- [8] Sakellariou, R. ; Zhao, H. : “A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems” , *IPDPS (2004)*.

- [9] Lin, M. ; Ku, K. : “The Distributed Program Reliability Analysis on Star Topologies” , *IEICS Trans. Information and Systems* (1998), E82-D, 1020-1029.
- [10] Raghavendra, C.S. ; Prasanna, V.K.; Hariri, S. :“Reliability Analysis in Distributed Systems”, *IEEE Trans. Computing* (1988), Vol.37, No.3.
- [11] Shatz, S. M. ; Wang : “Task Allocation for Maximizing Reliability of Distributed Computing Systems”, *IEEE Trans. Computing* (1992) ,41, 1156-1168.
- [12] Shatz, S. M. ; Wang : “Models&Algorithms for reliability oriented task-allocation in redundant distributed computer Systems”, *IEEE Trans. Reliab.*(1989) ,38,16-26.
- [13] Kartik, S. ; Murthy : “Task allocation Algorithms for maximizing reliability of distributed computing systems”, *IEEE Trans. Comput.*( 1997), 46, 719-724.
- [14] Qin, X. ; Jiang : “Dynamic ,reliability-driven scheduling of parallel real-time jobs in Heterogeneous Systems”, *In Proc. Int. Conf. Parallel Processing* (2001).
- [15] Hou, C. ; Shin, K.G. : “Allocation of Periodic Task Modules with Precedence and Deadline Constraints in Distributed, Real-Time Systems” , *IEEE Trans. Computing* (1997).
- [16] Juedes, D. ; Drews, F. : “Heuristic Resource Allocation Algorithms for Maximizing Allowable Workload in Dynamic, Distributed Real-Time Systems” , *IPDPS* (2004).
- [17] Chen, G. ; Kandemir, M. ;Tosun,S.; Sezer, U.: “Reliability-Conscious Process Scheduling under Performance Constraints in FPGA-Based Embedded Systems”, *IPDPS* (2005).
- [18] Tosun, S. ;Mansouri, N. ; ; Arvas, E.; Kandemir, M.;Xie: “Reliability-Centric High-Level Synthesis”, *DATE* (2005).
- [19] Tosun, S. ;Mansouri, N. ; Arvas, E.; Kandemir, M.;Xie,Y.;Hung,W-L. : “Reliability-Centric Hardware/Software Codesign”, *ISQED* (2005).
- [20] Tosun, S. ; Ozturk, O. ; Mansouri, N.; Arvas, E.; Kandemir, M.;Xie,Y.;Hung,W-L. : “An ILP Formulation for Reliability-Oriented High-Level Synthesis”, *ISQED* (2005).
- [21] Patvardan, C. ; Prasad, C.C.; Pyara, V.P. : “Generating of K-Trees of Undirected Graphs”, *IEEE Trans. on Reliability* (1997), Vol.46,No.2.

- [22] Page, A.J. ; Naughton, T.J. : “Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing” *IPDPS (2005)*.
- [23] Dogan, A ; Ozguner, F. : “Matching and Scheduling Algorithms for Minimizing Execution Time and Failure Probability of Applications in Heterogeneous Computing”, *IEEE Trans. Parallel and Distributed Systems (2002)*, Vol.13, No.3.
- [24] Doğan, A ; Özgüner, F : “Biobjective Scheduling Algorithms For Execution Time-Reliability Trade-Off in heterogeneous Computing Systems” *The Computer Journal (2005)*, Vol. 48, No.3.
- [25] Doğan, A ; Özgüner, F. : “Matching and Scheduling Algorithms for minimizing execution time and failure probability of applications in heterogeneous computing”, *IEEE Trans. Paralle. Distr. (2002)*, 13, 308-323.
- [26] Topcuoglu, H. ; Demiroz, B. : “Static Task Scheduling with a Unified Objective on Time and Resource Domains” , *Computer Journal (2006)*
- [27] Atakan Dogan: “Matching and Scheduling of Applications in Heterogeneous Computing Systems with Emphasis on High-Performance, Reliability and QoS”, PhD thesis, Ohio State University, Columbus, OH. (2001).
- [28] Darrel Whitley: “The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best”, *Proceedings of the 3rd International Conference on Genetic Algorithms (1989)*.
- [29] Eiben, A.E.; Smith, J.E. : “Introduction to Evolutionary Computing” *Springer Publication, (2003)*.
- [30] Sugavanam, P. ; Siegel, H.J. ; Maciejewski, A.; Oltikar, M.; Mehta, A.; Pichel, R.; Horiuchi, A.; Shestak, V.; Al-Otaibi, M.; Krishnamurthy, Y.; Ali, S.; Zhang, J.; Aydin, M. Lee, P. Guru, K. Raskey, M. Pippin, A.: “Robust Static Allocation of Resources for Independent Tasks under Makespan and Dollar Cost Constraints”, *JPDC (2005)*.
- [31] Correa, R. C. ; Ferreira, A. ; Rebreyend, P. : “Integrating list heuristics into genetic algorithms for multiprocessor scheduling”, *IEEE SPDP (1996)*.
- [32] Correa, R. C. ; Ferreira, A. ; Rebreyend, P. : “Scheduling Multiprocessor tasks with Genetic Algorithms”, *IEEE Trans. Parallel and Distributed Systems (1999)*, Vol.10, No.8.

- [33] Wang, L. ; Siegel, H. J. ; Roychowdhury, V. P. : “A Genetic Algorithm Based Approach for Task Matching and Scheduling”, *IEEE HCW (1996)*.
- [34] Zomaya, A. ; Ward, C. ;Macey, B. : “Genetic Scheduling for Paralel Processor Systems: Comparative Studies and Performance Issues”, *IEEE Trans. On Paralel and Distributed Systems (1999)*.
- [35] Betül Demiröz: “Hybrid Evolutionary Algorithms for Solving the Register Allocation Problem” *Master Thesis, Marmara University, Istanbul, Turkey (2004)*.

# ESMA YILDIRIM

## **OBJECTIVES**

---

*Academically improve my computer skills*

## **EXPERIENCE**

---

2005–2006    Fatih University  
İstanbul  
*Lecturer*

2004-2005    Avrupa Yazılım  
  
İstanbul  
*Software Engineer*

## **EDUCATION**

---

2004–Present    Marmara University (MS in Computer  
Engineering)  
İstanbul

2000–2004    Fatih University (BS in Computer Engineering)  
İstanbul

## **RESEARCH INTERESTS**

---

Java technologies, Distributed Systems, Task Scheduling,  
Software Engineering, Evolutionary Algorithms

## **PROFESSIONAL SKILLS**

---

English(Fluent), German(beginner),Arabic(Intermediate)