

**T.C.
MARMARA UNIVERSITY
INSTITUTE FOR GRADUATE STUDIES IN
PURE AND APPLIED SCIENCES**

**LOCALITY AWARE TASK SCHEDULING IN
HETEROGENEOUS COMPUTING
ENVIRONMENTS**

**Alper Köse
(Computer Engineering)**

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

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

İSTANBUL 2007

**T.C.
MARMARA UNIVERSITY
INSTITUTE FOR GRADUATE STUDIES IN
PURE AND APPLIED SCIENCES**

**LOCALITY AWARE TASK SCHEDULING IN
HETEROGENEOUS COMPUTING
ENVIRONMENTS**

Alper KÖSE
(Computer Engineer)
(141100320040251)

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

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

İSTANBUL 2007

MARMARA UNIVERSITY
THE INSTITUTE FOR
GRADUATE STUDIES IN PURE AND APPLIED SCIENCES

ACCEPTANCE AND APPROVAL DOCUMENT

**LOCALITY AWARE TASK SCHEDULING IN HETEROGENEOUS
COMPUTING ENVIRONMENTS**

Established committee listed below, on and by the *INSTITUTE FOR GRADUATE STUDIES IN PURE AND APPLIED SCIENCES*' Executive Committee, have accepted Mr. Alper KÖSE's Master of Science thesis, titled as "Locality Aware Task Scheduling in Heterogeneous Computing Environments" in Computer Engineering Programme.

COMMITTEE

Advisor : Assoc. Prof. Dr. Haluk TOPÇUOĞLU, Marmara University
Member : Prof. Dr. Fuat İNCE, Marmara University
Member : Asst. Prof. Dr. Serol Bulkan, Marmara University

Date of thesis' / dissertation's defense before the committee :

.....

APPROVAL

Mr. Alper KÖSE has satisfactorily completed the requirements for the degree of Master of Science in Computer Engineering at Marmara University. Mr. Alper KÖSE is eligible to have the degree awarded at our convocation on Diploma and transcripts so noted will be available after that date.

Istanbul

DIRECTOR

ACKNOWLEDGEMENT

It is a pleasure to thank the all people who made this thesis possible.

I would like to thank to my thesis supervisor Assoc. Prof. Dr. Haluk Topçuoğlu for his endless support, encouragement and guidance. I would like to thank Prof. Dr. Fuat İnce and Asst. Prof. Dr. Serol Bulkan for participating in my thesis committee and for their revisions on the thesis.

I also would like to special thanks to my colleagues in Siemens, Caner Akpınar, Hüseyin Gazi Ozkanat and Serdal Bayram, for their valuable support and to Fatma Engin Corut for her technical support.

I am grateful to my family and my beloved one, Elif Sürer, for their endless love and support.

October, 2007

Alper KÖSE

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGEMENT.....	I
TABLE OF CONTENTS	II
ÖZET	V
ABSTRACT	VI
CLAIM OF ORIGINALITY	VII
ABBREVIATIONS	VIII
LIST OF FIGURES	IX
LIST OF TABLES.....	XI
PART I. INTRODUCTION.....	1
I.1 CONTRIBUTIONS OF THE THESIS	4
I.2 OUTLINE OF THESIS	4

PART II. PROBLEM DEFINITION	5
II.1. LOCALITY AND DATA REUSE CONCEPTS	7
II.2. RELATED WORK	9
II.2.1. HEFT Algorithm.....	11
II.2.2. DLS Algorithm.....	11
II.2.3. Locality-Aware Scheduling.....	12
PART III. GENETIC ALGORITHM	15
III.1. COMPONENTS OF GENETIC ALGORITHM	16
III.1.1. String Representation	16
III.1.2. Initialization	17
III.1.3. Parent Selection	17
III.1.4. Crossover and Mutation	18
III.1.5. Survivor Selection	19
III.1.6. Termination Condition	19
III.2. HYBRID GENETIC ALGORITHMS.....	19
PART IV. SCHEDULING ALGORITHMS	22
IV.1. LHEFT ALGORITHM.....	22
IV.2. HYBRID GENETIC ALGORITHM FOR LOCALITY AWARE SCHEDULING.....	24
IV.2.1. Initial population generation	24
IV.2.2. Genetic operators.....	25
IV.2.2.1. Parent Selection.....	25
IV.2.2.2. Crossover	25
IV.2.2.3. Mutation.....	27
IV.2.3. Local Search.....	27
IV.2.4. Survivor Selection	28
IV.2.5. Termination Condition	28

PART V. EXPERIMENTAL STUDY.....	30
V.1. PERFORMANCE METRICS	30
V.2. GENERATING RANDOM TASK GRAPHS	31
V.3. EXPERIMENTS FOR IDENTIFYING GA PARAMS	33
V.4. COMPARISON STUDY BASED ON RANDOMLY GENERATED TASK GRAPHS	35
V.4.1. Comparison with respect to Graph Size	36
V.4.2. Comparison with respect to CCR.....	37
V.4.3. Comparison with respect to Locality	40
PART VI. CONCLUSION.....	43
REFERENCES	45
RESUME	49

ÖZET

HETEROJEN HESAPLAMA ORTAMLARINDA YERSELLİK BİLİNÇLİ GÖREV ÇİZELGELEME

Heterojen hesaplama ortamları için etkili görev çizelgeleme “NP-Complete” olan bir problemdir ve genel görev çizelgeleme probleminin çözümü için literatürde birçok algoritma önerilmiştir. Görev çizelgelemeye ek olarak önbelleklerin kullanımı da performans ve güç tüketimi açısından kritik bir öneme sahiptir. Literatürdeki ilgili çalışmalar, önbellek kullanımını etkin kılmayı genellikle donanım ya da derleme seviyesinde hedeflemişlerdir. Önbellek davranışlarının Görev seviyesinde optimize edilmesinin hedeflenmesi iyi bir katkı olabilir. Çalışmamızda öncelikle veriye bağımlılıkları ve tekrar veri kullanımını göz önüne alarak oluşturduğumuz yersellik bilinçli modeli tanıtıyoruz. Daha sonra bu modeli temel alarak Yersellik-Bilinçli Heterojen Erken Bitiş Zamanı (LHEFT) ve Melez Genetik Algoritma (HGA) olarak iki algoritma öneriyoruz. Önerilen algoritmaların performansları, sentetik olarak oluşturulmuş görev çizgeleri kullanılarak literatürde önde gelen üç algoritma ile karşılaştırıldı. Bu çalışmamızda sunduğumuz karşılaştırma çalışmasına göre melez genetik algoritmamız tüm performans göstergelerinde diğer tüm ilgili çalışmalardan daha iyi sonuç göstermiştir.

Anahtar Kelimeler: Görev Çizelgeleme, Heterojen Hesaplama Ortamları, Yersellik, Heuristik, Genetik Algoritmalar

Ekim, 2007

Alper KÖSE

ABSTRACT

LOCALITY AWARE TASK SCHEDULING IN HETEROGENEOUS COMPUTING ENVIRONMENTS

Efficient task scheduling in heterogeneous computing environments is an NP-complete problem and there are many algorithms proposed in the literature so far to solve various types of the general task scheduling problem. In addition to task scheduling decisions, utilizing caches is also critical for both performance and power perspectives. Related work in the literature usually target for optimizing caches at hardware level or compilation level. Targeting task level optimizations for cache behaviors could be good asset and it can be performed at application level. In this study, we first present a locality aware task scheduling model by considering data dependencies and data reuses between tasks. Additionally two locality aware task scheduling heuristics are proposed in this thesis based on the proposed model, which are Locality-aware Heterogeneous Earliest Finish Time (LHEFT) algorithm and a Hybrid Genetic Algorithm (HGA). The performance of the proposed algorithms is compared with the three leading scheduling algorithms in the literature by using synthetically generated task graphs. According to comparative study presented in this study, our hybrid genetic algorithm outperforms the related work with respect to all performance metrics considered in this study.

Keywords: Task Scheduling, Heterogeneous Computing Environment, Locality, Heuristics, Genetic Algorithms

October, 2007

Alper KÖSE

CLAIM FOR ORIGINALITY

LOCALITY AWARE TASK SCHEDULING IN HETEROGENEOUS COMPUTING ENVIRONMENTS

A new model for exploiting the locality in task scheduling of a given application for an heterogeneous computing environment is presented in this study. Additionally, two new algorithms, which are Locality-Aware Heterogeneous Earliest Finish Time (LHEFT) Algorithm and a Hybrid Evolutionary Algorithm, are presented by considering the proposed model as part of this study.

October, 2007

Assoc. Prof. Dr. Haluk TOPÇUOĞLU

Alper KÖSE

ABBREVIATIONS

DAG	:	Direct Acyclic Graph
drr	:	Data Reuse Rate
dft	:	Data Fetch Time
DL	:	Data Load time
HEFT	:	Heuristic Execution Finish Time
LHEFT	:	Locality-aware Heuristic Execution Finish Time
DLS	:	Dynamic Level Scheduling
DynL	:	Dynamic Level
LS	:	Locality-aware Process Scheduling
LMT	:	Levelized Min. Time
EFT	:	Execution Finish Time
EST	:	Execution Start Time
AFT	:	Actual Finish Time
AST	:	Actual Start Time
v	:	Number of tasks in graph
ρ	:	Number of processors
α	:	Shape of the graph
θ_{OUT}	:	Outdegree of nodes
ϕ_T	:	Task heterogeneity
ϕ_M	:	Machine heterogeneity
CCR	:	Communication to computation ratio
δ_{data}	:	Task-data dependency rate
δ_{drr}	:	Reuse rate

LIST OF FIGURES

FIGURE	PAGE NUMBER
Figure II.1 DAG of an Application with Communication and Computation Costs.....	6
Figure II.2 Sample Cache Snapshot	8
Figure II.3 The HEFT Algorithm	11
Figure II.4 DLS Algorithm.....	12
Figure II.5 LS Algorithm	13
Figure II.6 Sample Output Schedules of Algorithms.....	14
Figure III.1 General Pseudocode for GA	15
Figure III.2 Execution Flow of GA	16
Figure III.3 Two Sample Chromosomes for the Task Graph in Figure II.1	17
Figure III.4 Execution Flow of Simple Hybrid GA	20
Figure IV.1 Sample Output Schedule of LHEFT algorithm for task graph given in Figure II-1	23
Figure IV.2 Example for Updates After Insertion in LHEFT	24
Figure IV.3 Applying order-based crossover operator	26
Figure IV.4 Coupling Pseudo Code.....	26
Figure IV.5 Illustrations of mutation operators.....	27
Figure IV.6 Sample Output Schedule for GA.....	29
Figure V.1 Comparison of Crossover Types Without any Other Genetic Operator.....	33

Figure V.2 Effect of Data Amount and Data Reuse with CCR=0.1	40
Figure V.3 Effect of Data Amount and Data Reuse with CCR=1.0	41
Figure V.4 Effect of Data Amount and Data Reuse with CCR=10.0	42

LIST OF TABLES

TABLE	PAGE NUMBER
Table V-1 Comparison of Best Configuration for each Crossover and Mutation Type	34
Table V-2 Comparison of Local Search Methods for Crossover and Mutation Types	34
Table V-3 Comparison study of Population Coefficients.....	35
Table V-4 Comparison study of Iteration Coefficients.....	35
Table V-5 Comparisons of Algorithms with Respect to Graph Sizes for LoLo Heterogeneity Type	36
Table V-6 Comparisons of Algorithms with Respect to Graph Sizes for HiHi Heterogeneity Type	37
Table V-7 Comparisons of Algorithms with Respect to CCR values for LoLo Heterogeneity Type	38
Table V-8 Comparisons of Algorithms with Respect to CCR values for HiHi Heterogeneity Type	39

PART I

I. INTRODUCTION

Heterogeneous computing systems comprise growing numbers of diverse computing resources that can be local to one another or geographically distributed. The opportunity and need for effectively utilizing heterogeneous computing resources has given rise to the notions of cluster computing, grid computing, and peer-to-peer computing. Heterogeneous computing environments consist of diverse set of computing resources connected with a network. They provide a computing platform that can support executing computationally intensive parallel and distributed applications. The efficient scheduling of tasks of an application on the available resources is one of the key factors for achieving high performance.

The general task scheduling problem in a heterogeneous computing environment includes assigning the tasks of an application (which is represented by a directed acyclic graph) to processors in the environment and ordering the task executions on each processor in order to minimize overall completion time of the application. Task scheduling is an NP-Complete problem [1] including some of the restricted cases [2] such as scheduling tasks with one or two time units to two processors, or scheduling unit-time tasks to an arbitrary number of processors. Due to its NP-completeness nature, task scheduling problem and its various types are studied considerably in the literature and many algorithms and techniques are proposed in the literature [3,4,5,6,13,16]. There are two types of scheduling strategies: *static* and *dynamic*. In static scheduling strategy all scheduling decisions should be determined before start of an application, therefore characteristics of an application (execution times, task dependencies, communication amounts *etc.*) are assumed to be known a priori. In dynamic scheduling strategy a run-time scheduler is assigned to orchestrate execution at the time application is actually running [22], therefore knowledge of

characteristics of an application is not required at the beginning of a schedule. In task scheduling problems characteristics of an application is reflected using directed acyclic graphs (DAG). Nodes of a DAG represents application tasks and edges represents task dependencies. Weights of nodes shows expected computation costs and weights of edges shows expected communication costs. Objective of a task scheduling problem is to assign task onto computing resources by satisfying task-precedence requirements and obtaining minimum overall completion time.

In a heterogeneous computing environment, geographically distributed set of different machines interconnected with high-speed links are utilized to execute computationally-intensive applications with diverse requirements. Due to locations of machines, data transfer between machines may take long time and lengthens overall completion time. Since several tasks of an application may require (partially or fully) same data sets, it will be beneficial to exploit locality decisions during task scheduling phase. Even task-dependency communication costs are at low levels, data transfer rates may become a bottleneck if a task has to fetch considerable amount of data. Exploiting locality may be very helpful in these circumstances. Simple example can be an application for annual reports of a bank.

Locality-aware compiler optimization techniques in the literature mainly focus on loop and data-layout optimizations. Since hardware technology has improved and multi-core environments have become widespread, locality optimizations are gaining importance. In a recent study [12] that focuses on multiprocessor system on a chip (MPSoCs) they design a scheduler that retrieves data sharing information between processes and use this information for scheduling the processes on multiple cores. In order to increase the use of locality and to decrease cache misses, data layout optimizations are also considered.

On-chip memory performance is a significant parameter in embedded multiprocessor system-on-a-chip (MPSoC) based systems since inappropriate memory utilization may cause lots of accesses that degrade the performance. Since utilization of memory depends on both individual processes and interaction among different processes, the performance utilization issue should not be limited to the individual accesses. To maximize the performance, two possible solutions are proposed. First, the processes which do not share data can be scheduled at different cores so that the number of conflicts decrease. Secondly, the processes that share data but that are required to work at different times can work on the same processor [12].

The work of Kandemir and Chen [12] aims to design a scheduler that retrieves data sharing information between processes and to use the information while scheduling

the processes on multiple cores. To increase use of locality and decrease cache misses data layout optimizations are used.

In another work [24], a compiler-driven approach is proposed that aims to decrease off-chip memory references in multi-processor SoCs. If the compiler predicts whether data that has already been used by the current processor will be used by another processor after a short-time, then caching data in local memory of the current processor will be a good approach.

To get benefit from this approach, caching mechanism should work in a feasible manner. Such a feasible method will lead the next processor - the one that will use the data - to fetch the data from the local memory of the current processor. This will prevent access to an off chip reference, so the number off-chip references will decrease.

Both of these studies inspect code of the application first in compile-time and either optimize it using loop transformations in compile-time or use data-layout optimizations in running time in order to enhance locality. Apart from these compiler-level studies, there is no application-level task scheduling framework that considers locality of data for tasks of a given application.

The aim of this thesis is to apply locality conscious methods to heterogeneous computing environments. In this thesis, we first present a locality aware task scheduling model for heterogeneous computing environments by considering data dependencies and data reuses between tasks. Then, this new model is formulated with two new algorithms, which are an algorithm called Locality-aware Heterogeneous Earliest Finish Time (LHEFT) and an Hybrid Genetic Algorithm. The first algorithm is the extended version of the HEFT algorithm [3], which is a task scheduling algorithm for heterogeneous processors that surpasses the previous approaches in terms of both quality of schedules and the running time for generating the output schedules. In the LHEFT algorithm, upward rank definition is modified to include locality measurements; and the processor selection phase is modified according to the new model, which are presented in Part 4. The second algorithm is an Hybrid Genetic Algorithm.

I.1 CONTRIBUTIONS OF THE THESIS

Main contributions of this thesis are as follows:

- Locality concepts are not used in heterogeneous environments. Although recent studies try to exploit data locality in MPSoCs, we couldn't find any study to apply this concept on heterogeneous environments.
- As part of task scheduling problem, we define locality in terms of both data-reuse and task-data dependency.
- We propose a hybrid genetic algorithm that performs outstanding performance on most of the experiments.
- Local search operator tries to minimize execution costs of the tasks in critical path considering locality improvements.
- We extend well known HEFT algorithm in order to exploit locality. HEFT algorithm performs very good on heterogeneous computing environments. Extended LHEFT algorithm has all advantages of HEFT algorithm, and using the knowledge of locality displays very good performance on experiments.

I.2 OUTLINE OF THESIS

The rest of this thesis is organized as follows. In Part 2 we discuss the general task scheduling problem and give details about locality oriented task-scheduling. We also represent brief information on related work in the literature. Last section discusses general aspects of genetic algorithms. Part 3 presents the details of two algorithms proposed, which are. Locality-aware Heterogeneous Earliest Finish Time (LHEFT), and Hybrid Genetic Algorithm (HGA). Experimental results of implemented algorithms and comparison of reference algorithms is presented in Part 4. Our experiments are based on synthetically generated graphs. In order to obtain these graphs we implemented a random graph generator that generates task graphs, execution and locality costs, based on given characterization parameters. Then a set of experiments are presented in order to set good values to all parameters of HGA. Last section of this chapter gives details about our experiments of performance metrics on proposed algorithms. Finally, in Part 5 conclusion and future work is presented.

PART II

II. PROBLEM DEFINITION

In this thesis, we extend the system and application models presented in the literature[3] with locality based issues. As described in Part I, static task scheduling model is used, when the characteristics of an application (execution times, task dependencies, communication amounts *etc.*) are known a priori. In static task scheduling, applications are represented by *directed acyclic graphs* (DAGs). $G = (V, E)$ is a DAG where V is the set of v tasks (nodes), and E is the set of e communications between tasks (edges). Each edge (i, j) represents the precedence constraint between task t_i and t_j where task t_i should complete its execution before task t_j starts. In a DAG, a task that does not have any parent is called an entry task and a task without a child is called an exit task. If there is more than one entry task or exit task, those tasks are connected to a pseudo-entry task (source) or pseudo-exit task (sink), respectively, with no communication cost and no computation cost of pseudo tasks. Thus, schedule is not affected.

Our system and application models are based on a previous work [3]. We assume that target computing environment consists of q heterogeneous processing elements in a fully connected topology and all inter-processor communications occur without contention. Execution of tasks is assumed to be non-preemptive and sequential *i.e.* no parallelism within a task). W is a $v \times q$ computation cost matrix in which $w_{i,k}$ gives estimated execution time to complete task t_i on processor p_k . A task cannot begin

execution until it has received all its inputs and cannot give any output until it has completed execution. At the end of the execution, all outputs are simultaneously ready for communication. All tasks' execution times on each processor and their communication costs are known *a priori*.

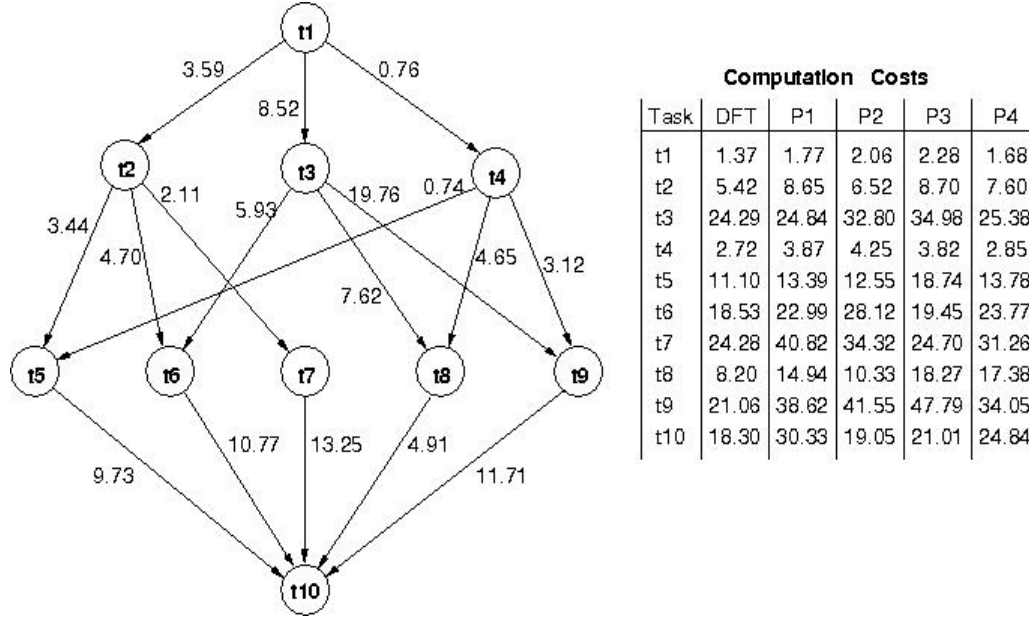


Figure II.1 DAG of an Application with Communication and Computation Costs

For an edge (i, j) of DAG, if t_i and t_j are mapped into same processor, then communication cost becomes negligible compared to interprocessor communications, so we assume it is 0. Assuming that t_i and t_j are mapped on p_m and p_n , the communication cost for the edge (i, j) is defined by,

$$c_{i,j} = L_m + \frac{commData_{i,j}}{B_{m,n}}, \quad (II.1)$$

where L is a q -dimensional vector containing communication startup costs of processors, B is a $q \times q$ matrix of data transfer rates between processors and $commData_{i,j}$ is the amount of data to transfer from t_i to t_j .

$EST(i, k)$ is the earliest execution start time and $EFT(i, k)$ is the earliest execution finish time of task t_i on processor p_k . Defined by

$$EST(i, k) = \max \left\{ avail[k], \max_{t_m \in pred(i)} (AFT(m) + c_{m,i}) \right\}, \quad (II.2)$$

$$EFT(i, k) = w_{i,k} + EST(i, k), \quad (II.3)$$

where $pred(i)$ is the set of predecessor tasks of t_i and $avail[k]$ is the earliest available time at which processor p_k is ready for task execution. Inner *max* block in the equation II.3 returns the time when all inputs needed for execution of task t_i on processor p_k have arrived and t_i is ready for execution. $AFT(m)$ is the actual finish time of task t_m . Since there is no predecessor of an entry task, we can shorten the equation:

$$EST(t_{entry}, k) = 0, \quad (II.4)$$

Objective in static scheduling is to assign tasks of DAG to processing elements such that the schedule length (also known as *makespan*) is minimized without violating the precedence constraints.

$$makespan = AFT(t_{exit}), \quad (II.5)$$

II.1. LOCALITY AND DATA REUSE CONCEPTS

Locality shows similarities of memory accesses between tasks. It reflects the level of concentration of the accesses to the same memory during execution [8]. Locality can affect execution time if two tasks, executing on the same processor, have to load data used by both tasks [12]. Assume that data requirements of two tasks t_j and t_i partially (or fully) overlapped (as given in Figure II.2); and assume that task t_j has already scheduled on processor p_k and its data is loaded into memory. After the completion of task t_j , if data fetched for t_j is already in the memory, then locality-aware scheduling of t_i on the same processor will reduce the completion time of task t_i ; which is due to reducing the cache misses and page faults.

Locality is represented by the amount of *data reuse*, which has two components. First one is how much data is needed for executing a given task, which is called *task-data dependency*. If a task fetches high amounts of data for execution, then execution time deteriorates due to cache misses and costly memory accesses. If a task fetches low amounts of data, then its effect may not be that much. This component is task specific and machine independent. We define dft_i as required time for task t_i to fetch its data (*i.e.* data fetch time). We assume that communication bandwidth for a

memory access is homogeneous across the environment. Since bandwidth is homogeneous across the environment, dft_i is proportional to fetched data's size.

Second component of locality is the *data reuse rate*, which denotes percentage of the task's data already pre-fetched by the previously executed task on the same processor. Data reuse rate is denoted by $drr_{i,j}$ where task t_i will execute after task t_j . Assume that, task t_i and t_j are mapped onto same processor, and $data_i$ is equal to 600MB and $data_j$ is equal to 400MB. t_i will execute after t_j , so t_i will use some of the data fetched by t_j . If $drr_{i,j}$ is 0.5 (50%), then amount of data shared between these two tasks is 300MB (i.e. t_j used 400MB data and 300MB of t_j 's data is used by t_i). Therefore, instead of loading 600MB data, t_i uses 50% of its data from cache). If there is no precedence constraint between these two tasks and execution order is reversed, amount of shared data will not be changed but data reuse rate will be different; i.e., the $drr_{j,i}$ will become 0.75 (75%).

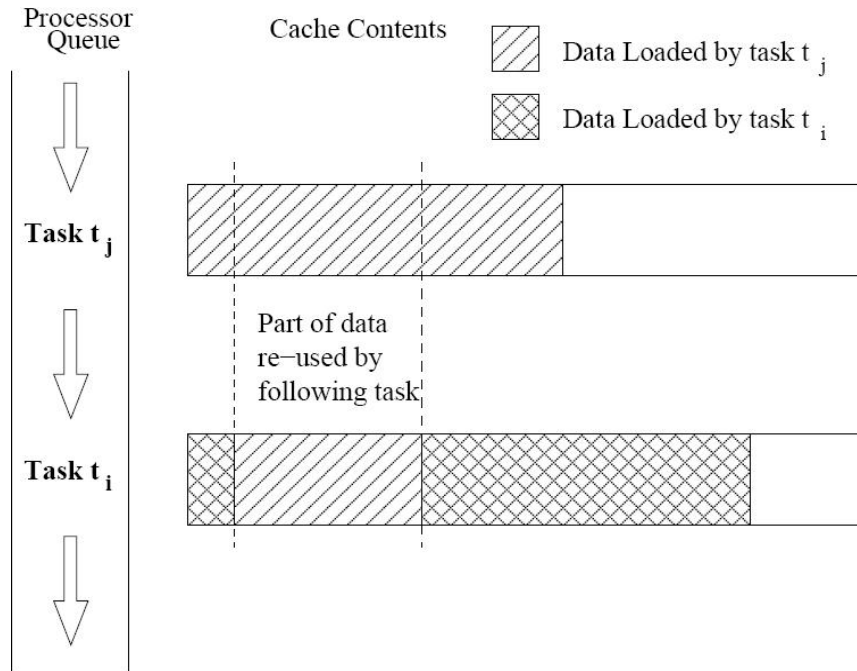


Figure II.2 Sample Cache Snapshot

These relations are shown in the formula:

$$T_{DATA_REUSE} = drr_{i,j} \times dft_i = drr_{j,i} \times dft_j, \quad (II.6)$$

Since data fetch time is proportional to the amount of data, we use dft value to denote time. If data reuse rates are high on average, then the effect of locality on

execution times becomes higher. Data reuse rate component seems similar to task-data dependency component but it is not. Firstly, by definition, it depends on another process. It is also dependent on the order of tasks. Amount of shared data remains the same but rates are different for each task.

In our model, $w_{i,m}$ is the execution time of a task t_i on processor p_m . Execution time of a task consists of two components: computation time and data loading time. We may formulate it like:

$$w_{i,m} = w'_{i,m} + DL(i, m), \quad (II.7)$$

where $w'_{i,m}$ is the computation time of CPU and $DL(i, m)$ returns data loading time for task t_i on processor p_m . Here $DL(i, m)$ can be defined as

$$DL(i, m) = dft_i - drr_{i, prev(m)} \times dft_i, \quad (II.8)$$

where $prev(m)$ is the previous task executed just before t_i on p_m . Therefore equation (II.7) will be considered to set computation costs given in earliest finish time given in Equation (II.3).

II.2. RELATED WORK

In this subsection, we briefly summarize related work for task scheduling and locality. Task scheduling is an NP-Complete problem [1] where optimal solutions can be found only after an exhaustive search, even in some restricted or limited cases [2] such as scheduling tasks with one or two time units to two processors, or scheduling unit-time tasks to an arbitrary number of processors. Task scheduling with efficiency in performance is very important in heterogeneous computing environments. Because of this importance, task scheduling is studied considerably and various algorithms proposed in the literature [3, 4, 5, 13, 16, 6, 7]. In one of the previous work [3] static task scheduling algorithms are classified into categories such as list scheduling algorithms, clustering algorithms, task duplication based algorithms, and guided random search algorithms.

List scheduling [3,4,5] algorithms construct a list of tasks by calculating each task's priority. Using priority based list, tasks are selected in that order and assigned to a processor. List scheduling algorithms have lower complexity and generally give good results.

Clustering algorithms [26,27,28,29] have three phases. In the first phase tasks are grouped into a set of unbounded number of clusters, in the second phase clusters are mapped into a set of available processors, and in the third phase clusters are merged and ordered to work on each assigned processor. They are generally for unbounded number of processors so they may not be directly applicable. Some examples in this group are Dominant Sequence Clustering (DSC) [26], Linear Clustering Method [27], Mobility Directed [33], and Clustering Scheduling System (CASS) [28].

Task duplication based algorithms are based on redundantly assigning tasks in order to reduce interprocessor communication overhead. The algorithms in this group are usually for an unbounded number identical processors and have higher complexity values relative to the other groups. As an example, BTD [31] algorithm and the DSH [30] algorithm have $O(v^4)$ complexity; the complexity of the CPFD [32] algorithm is $O(e \times v^2)$ for scheduling v tasks with e edges on a set of homogeneous processors.

Guided random search algorithms use random choices to guide iterations through problem space. These techniques use knowledge gained from previous steps and blend it with randomness to create a new solution. Guided random search algorithms mostly consist of genetic algorithm applications [16, 20, 7]. GAs provide good solutions, but they have much higher execution times than other heuristic techniques.

None of the methods presented above concentrate on data reuse. On the other hand, compiler-oriented studies mostly consider locality in their frameworks. They use locality to improve code structures on compile time. These studies mostly focus on optimizing program instances. Since hardware technology has improved and multi-core environments have become widespread, locality optimizations are gaining importance. Some recent studies consider using data locality in OS level task scheduling [12]. There is not much work that uses data localities in heterogeneous environments. However, these relatively low level scheduling studies may provide insight into our study.

In our comparison study, we measure the performance of our algorithms by comparing with three leading algorithms from the literature, which are Heterogeneous Earliest-Finish-Time (HEFT) [3], Dynamic Level Scheduling (DLS) [4] and Locality Aware Process Scheduling (LS) [12]. Here we present a summary of these algorithms.

II.2.1. HEFT Algorithm

The Heterogeneous Earliest Finish Time (HEFT) algorithm [3] is a list scheduling algorithm. Tasks are prioritized based on upward ranking, where the upward rank of a task t_i is defined by

$$rank_u(t_i) = \bar{w}_i + \max_{t_j \in succ(t_i)} (\bar{c}_{i,j} + rank_u(t_j)), \quad (II.9)$$

1. Set the computation costs of tasks and communication costs of edges with mean values
2. Compute $rank_u$ for all tasks by traversing graph upward, starting from the exit task
3. **while** there are unscheduled tasks in list **do**
4. Select the first task, t_i from the list for scheduling
5. **for all** processor p_k **do**
6. Compute $EFT(t_i, p_k)$ value using insertion based scheduling policy
7. **end for**
8. Assign task t_i to the processor p_k that minimizes EFT of the task t_i

Figure II.3 The HEFT Algorithm [3]

In this equation, $succ(t_i)$ is the set of tasks that are successors (also known as children) of task t_i , $\bar{w}_i$ is the average computation cost of task t_i and $\bar{c}_{i,j}$ is the average communication cost from task t_i to t_j . The HEFT algorithm outperforms related work with respect to both schedule time and running time and it keeps this performance over wide range of graph types [3].

II.2.2. DLS Algorithm

Dynamic Level Scheduling [4] (DLS) algorithm sets computation cost of each task with the median value of computation costs. In each iteration DLS selects best task-processor pair from the ready queue of tasks that maximizes dynamic level value, which is defined by:

$$DynL(t_i, p_k) = rank_u^s(t_i) - EST(t_i, p_k), \quad (II.10)$$

where $rank_u^s(t_i)$ is the static upward rank, defined similar to equation (II.9) of HEFT algorithm but without communication costs. EST term is the earliest start time of the given task and the processor pair. For the heterogeneous environments, a new term was added to represent the difference between the task's median execution time on

all processors and its execution time on the current processor. The general DLS algorithm has an $O(v^3 \times q)$ time complexity.

1. Set the computation costs of tasks with mean values
2. Compute $rank_u^s$ for all tasks by traversing graph upward, starting from the exit task
3. **while** there are unscheduled tasks in ready list
4. **for all** task t_i on ready list
5. **for all** processor p_k
6. Compute $DynL(t_i, p_k)$
7. **end for**
8. **end for**
9. Select a task-processor pair that gives maximum $DynL$ value and assign
10. **end while**

Figure II.4 DLS Algorithm [4]

II.2.3. Locality-Aware Scheduling

Locality-Aware Scheduling (LS) [12] algorithm considers data localities described with some similarities in this study. We divide locality into two parts: data-reuse ratio and task-data dependency. However, this study uses combined definition of these two aspects, simply as *data reuse*. Locality-aware scheduling is more compiler oriented. It proposes two algorithms one of which uses data-layout optimizations in compile time. Since our environment and problem definition do not consider compiler optimizations, we will implement the other algorithm that does not use data-layout optimizations. The main focus of the algorithm, which is a greedy heuristic one, is the maximization of data reuse. In the initial step, the tasks that are independent of each other are identified; and the ones that contain maximum data sharing are removed in an iterative approach. Then, the next process that maximizes the data reuse with the previous process of the core is chosen to be executed on that core.

When the algorithm given in Figure II.5 [12] considered, PS denotes the set of ready tasks, IN denotes the set of independent tasks, X denotes number of processors and M denotes data sharing matrix. The algorithm tries to map tasks that have minimum locality, in order to increase locality by leaving tasks which are sharing more data in the first loop. On the other hand, the algorithm simply maps task giving maximum locality with the previously assigned task in the second loop.

```

1.  $IN = \{p \mid p \in PS \text{ and } t \text{ does not depend on any process}\}$ 
2.  $PS = PS - IN$ 
3. while  $|IN| > X$ 
4.   select  $t \in IN$  such that  $\sum_{k \in IN} M[t][k]$  is minimized
5.    $IN = IN - \{t\}$ 
6.    $PS = PS + \{t\}$ 
7. end while
8. schedule the processes in  $IN$  on the processing elements
9. while  $PS$  is not empty
10.  for all processor  $p_x$ 
11.     $t =$  previously scheduled process on  $p_x$ 
12.    Select  $k \in PS$  such that  $M[t][k]$  maximized
13.     $PS = PS - \{t\}$ 
14.  end for
15. end while

```

Figure II.5 LS Algorithm [12]

When the task graph and corresponding communication and computation costs given in Figure II.1 is considered, the HEFT algorithm has a makespan of 65.40 (Figure II.6.a), the DLS algorithm has a makespan of 75.57 (Figure II.6.b) and the LS algorithm has a makespan of 116.16 (Figure II.6.c).

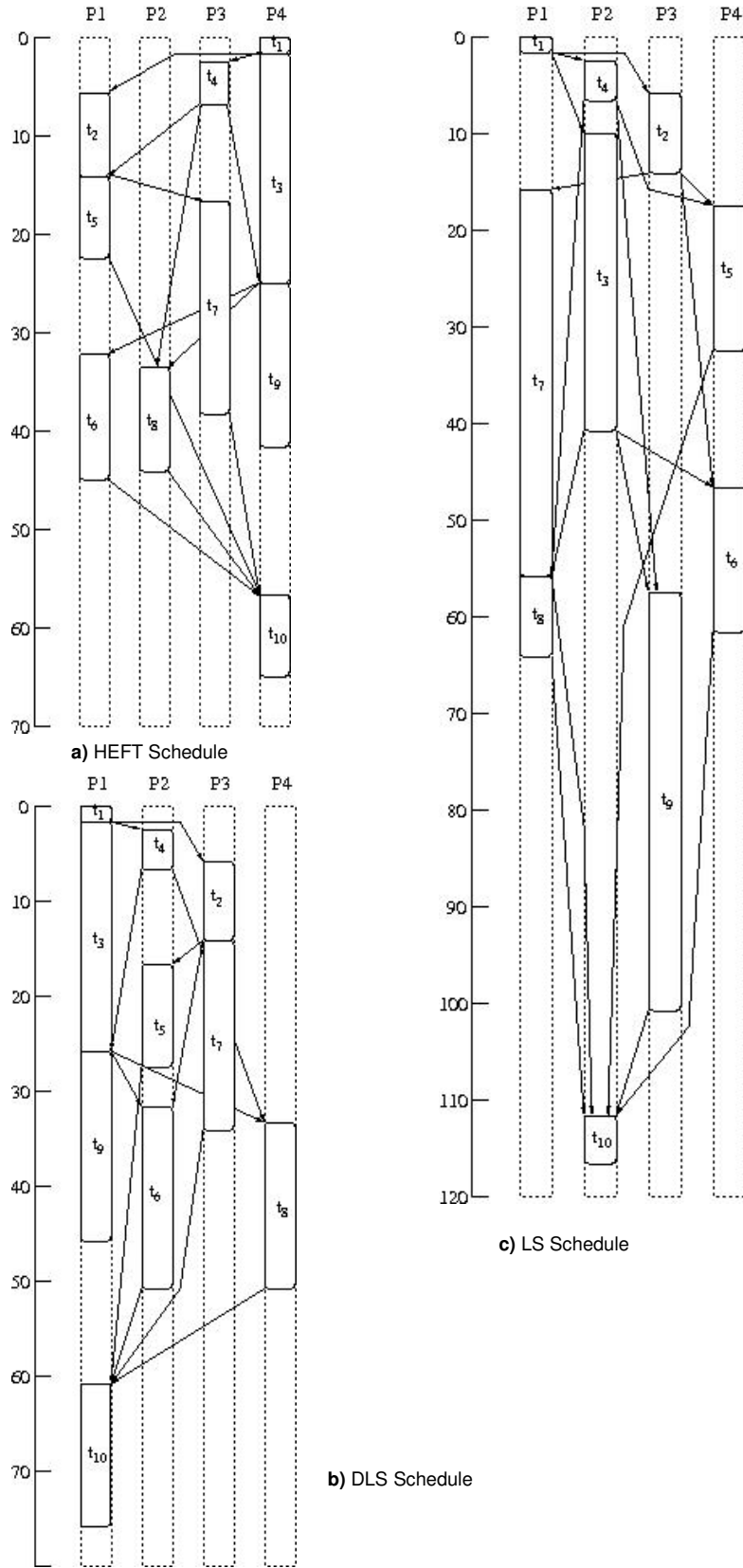


Figure II.6 Sample Output Schedules of Algorithms

PART III

III. GENETIC ALGORITHMS

Evolutionary Algorithms are heuristic search and optimization techniques which get inspiration from natural selection. Genetic Algorithms are the most popular and widely used class of Evolutionary algorithms, that is first developed by Holland in 1960's [21].

A genetic algorithm starts with an initial population, which is a set of solutions (*i.e.* candidate solutions), and then at each generation these individuals are evaluated with a problem specific fitness function. According to the defined selection scheme, individuals are selected to construct parent population. Genetic mutation and recombination operators are applied to generate new candidate solutions that forms the new population. After each generation, the algorithm checks the termination criteria. If termination condition has been reached, algorithm will stop. Figure III.2 shows the general schema of a Genetic Algorithm.

1. Initialize population
2. Evaluate the fitness of each individual in the population
3. **while** Termination condition is not satisfied
4. Select parents
5. Recombine pairs of parents
6. Mutate generated offsprings
7. Evaluate new offsprings
8. Select survivors for next generation
9. **end while**

Figure III.1 General Pseudocode for GA

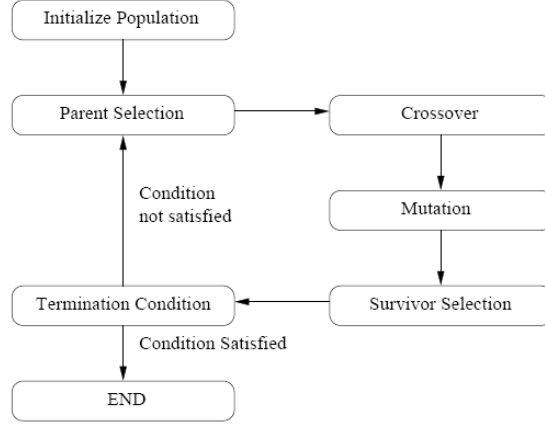


Figure III.2 Execution Flow of GA [9]

In order to work with GA, an *evaluation function* should be defined. Result of the evaluation function is the definition of improvement in GA. This evaluation function will compute *fitness value* of an individual. Although GA's general assumption is “higher the fitness better the solution”, since our objective function is to minimize the makespan we will reverse this rule: “lower the fitness, better the result”.

III.1. COMPONENTS OF GENETIC ALGORITHM

In this part we describe other major components of a genetic algorithm which are string representation, initialization, parent selection, crossover and mutation, survivor selection, and termination condition. Implementation details will be discussed in Part IV.

III.1.1. String Representation

There are various representations for GAs, but for task scheduling some specific representations are needed. Rebreyend, Sandnes and Megson [19] divide representations for GA-based task scheduling algorithms into two groups: *direct* and *indirect*. In direct representation, a chromosome gives only one solution and its representation always gives same makespan [16, 17, 20]. In an indirect representation, a decoding method must be used to get solution from a chromosome [14, 15, 18]. Thus, a chromosome may give different solutions considering decoding mechanism. We consider direct representation in our study. In order to represent a schedule we include task-processor mappings and order of execution of tasks. In our work, each chromosome is represented by a list of pairs with a length that is equal to number of tasks (nodes) in DAG. Each pair has two properties; a task and its

assigned processor, i.e., the i^{th} pair $pair_i = (t_j, p_k)$ in a chromosome denotes that task t_j is assigned to processor p_k . Ordering in chromosome preserves precedence constraints of DAG. It should be noted that if there is an edge between t_a and t_b in the task graph then t_a should come before t_b in chromosome. In Figure III.3 you can see two valid chromosomes for task graph in Figure II.1 are shown.

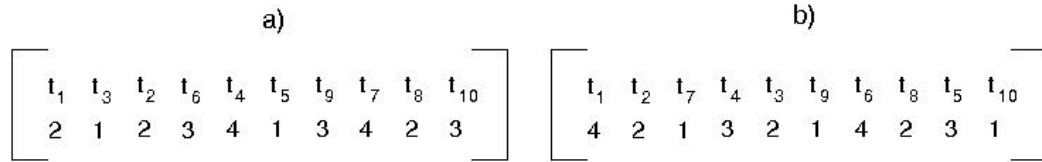


Figure III.3 Two Sample Chromosomes for the Task Graph in Figure II.1

III.1.2. Initialization

GA works on a set of solutions. At the beginning, this set must be created as the basis of population. This population is a small subset of problem solution domain. It contains various individuals such that each can be a valid solution to our problem. Purpose of random initialization is to cover solution space as much as possible. Sometimes initial population may be seeded by some algorithms in order to give direction to algorithm. If all the population is seeded, it may not be able to cover whole problem space and tends to converge to local minima.

III.1.3. Parent Selection

In GA, the role of parent selection is to generate offsprings from parents with good fitness value so that better individuals are chosen to seed the next generation. Together with the survivor selection mechanism, parent selection is responsible for pushing quality improvements [9].

Using probabilistic approach, individuals with good fitness values are more likely to be selected as parent than the ones with bad fitness values. However, always selecting best parents may cause premature convergence (i.e. population is dominated by similar individuals and get stuck in local minima thus preventing diversity and improvement).

There are various parent selection algorithms presented in the literature [9]. Most popular of them are roulette wheel and tournament selection. Roulette wheel selection is based on spinning a roulette wheel where selection probability of an individual is reflected as the sizes of the holes. It uses knowledge of the entire

population to create selection probabilities. Higher ranking fitness values have higher selection probabilities. On the other hand, tournament selection does not need the overall knowledge of the population. It selects some number of individuals (defined by the size of tournament) from population randomly. Then, it selects best ones from these individuals comparing their fitness values. Implementation of tournament selection is simple and less costly. Selection pressure is the intensity with which an environment tends to eliminate an individual, and thus its genes, or to give it an adaptive advantage (e.g. it controls the selection ratio of best individuals). The other advantage of tournament selection is that selection pressure can be adjusted by changing tournament size. If the tournament size is kept large, then weaker individuals are less likely to be selected.

III.1.4. Crossover and Mutation

From selected parents, a new offspring should be introduced into new population. New offspring can be produced using these two genetic operators: crossover and mutation. By using these operators, created offspring contains many characteristics of its parents.

Crossover combines parents' genes and produces one or two offspring depending on its type and implementation. The idea behind crossover is that the new offspring may be better than both of the parents if it takes the best characteristics from each of the parents. Crossover occurs during evolution according to a user-definable crossover probability. There are various crossover types such as one point crossover, n-point crossover and uniform crossover. One point crossover type is inspired from biological crossover. It cuts parents from a point and exchanges parts to create new offspring. N-point crossover is very similar to one point crossover, but it uses more than one point to cut parents. Uniform crossover decides (with some probability – known as the mixing ratio) which parent will contribute each of the gene values in the offspring chromosomes. This type does not use segments of parents, rather it decides which parent's gene for each location will be used.

Mutation is a unary genetic operator. It is applied to a single individual (possibly an offspring) and it randomly modifies some part of chromosome. Mutation is supposed to be random in order to create diversity in solutions. Purpose of mutation in GA is to avoid local minima by preventing all individuals from becoming similar to each other. Sometimes mutation is considered as the one and only operator doing whole search work as in Evolutionary Programming, or not used at all as in Genetic Programming in some evolutionary computing areas other than GA.

III.1.5. Survivor Selection

Aim of the survivor selection is to create population for next generation using parent population and offspring population [9]. Parent selection is responsible from increasing the quality of population. Since the size of population is kept constant, we have to select individuals in order to create a new generation. Although age based selection is used frequently, this selection is mostly fitness based. An example for fitness based selection is to select from the combined set of parent and offsprings, considering their fitness values. Another example may be replacing worst part of a parent population with individuals from offspring population. For an age based selection example we can select only from offspring population in order to eliminate older individuals (parents).

Elitism is a method used generally in age based selection. It protects some of the better individuals of the old generation from elimination in survival selection, thus it prevents loss of fittest individuals in population.

III.1.6. Termination Condition

Termination condition shows that the algorithm reaches some kind of level or limitation so that it must be stopped. The most common termination conditions are the followings:

- total number of generations has been reached
- allocated CPU time has elapsed
- fitness (best individual or over all population average) does not improves for a number of generations
- any combinations of these conditions

III.2. HYBRID GENETIC ALGORITHMS

In a hybrid genetic algorithm, local search is applied during GA in order to improve fitness values. No Free Lunch theorem [10,11] states that no algorithm can perform better on every problem, Genetic algorithms perform well on searching and approximating optimal solution areas in all range of problems from various domains. Problem specific algorithms may perform better than GA, for some of the problems. In order to fill this gap, hybrid genetic algorithms are proposed which can make use of both GA and problem specific algorithms.

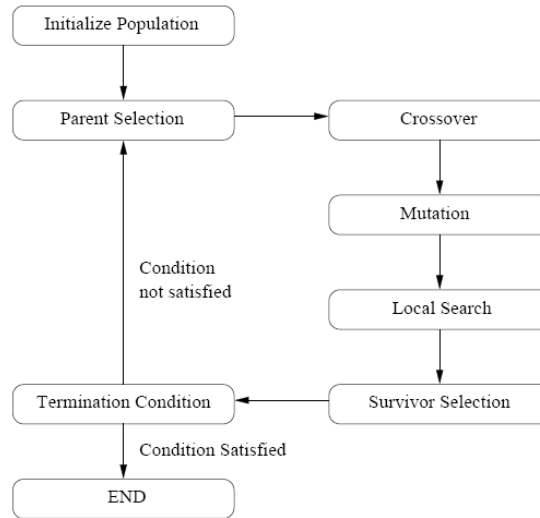


Figure III.4 Execution Flow of Simple Hybrid GA

The example shown in Figure III.4 is a simple hybrid genetic algorithm that uses local search factor as a fine-tuning tool for each individual. This is not the only way of hybridization; there are a number of ways that GA can make use of the local search methods [9]:

- *Heuristic or Intelligent Initialization:* In this way initial population used by GA is not created randomly as a whole. Some or all of the initial population consists of good individuals. This can be achieved by using one or more previously known good solutions to seed population. These improved individuals may be calculated with some other heuristics in the literature. Creating a larger initial population and selecting best individuals (*selective initialization*) may be an alternative approach. Another alternative approach is using local search for each individual and starting with locally optimal solutions of the search space.
- *Intelligent Crossover and Mutation:* Using problem specific knowledge in genetic operators is another hybridization method. You can inspect crossover point and if the outcome is not expected to give a good solution, another crossover point will be inspected. By definition of mutation, it should be random. Therefore, in hybridizing mutation, problem specific knowledge can be used to find areas to apply mutation, rather than outcome inspection.
- *Local Search on each individual:* This is a simple and a common way to implement hybrid techniques. In this method, local search is applied after genetic operators (crossover and/or mutation). Figure III.4 shows simple execution flow of this method. In our example there is one phase of local search, but there can be

more phases and they can be anywhere in the flow using output of genetic operators.

PART IV

IV. SCHEDULING ALGORITHMS

In this study we develop two algorithms that consider data reuse, which are Locality-aware Heterogeneous Earliest Finish Time (LHEFT) algorithm, and a hybrid genetic algorithm.

IV.1. LHEFT ALGORITHM

Locality-Aware Heterogeneous Earliest Finish Time (LHEFT) algorithm is based on HEFT algorithm [3] in the literature, where upward rank term in the HEFT algorithm is set according to the equation given below in order to consider locality measures:

$$rank_u(t_i) = \bar{w}_i - \sum_{t_j \in V} \frac{drr_{i,j} \times dft_i}{|V|} + \max_{t_j \in succ(t_i)} (\bar{c}_{i,j} + rank_u(t_j)), \quad (IV.1)$$

where $|V|$ is the total number of all tasks in set V . We use mean value of task t_i 's locality values since locality of task t_i will be effected by previously mapped task and it will affect locality of tasks mapped after itself. Therefore, if t_i has a high locality mean value, then it will not only decrease its execution time but also decrease execution time of task mapped afterwards. Note that as in the HEFT, task with the maximum rank value is selected at each step.

After task prioritizing phase, algorithm enters processor selection phase which maps tasks to processors. Using prioritized list of tasks, LHEFT algorithm tries to find an

appropriate processor that minimizes EFT value for each task. It should be noted that the EFT computation used in the HEFT algorithm does not consider locality. The LHEFT algorithm also uses EFT defined as in equation (II.3).

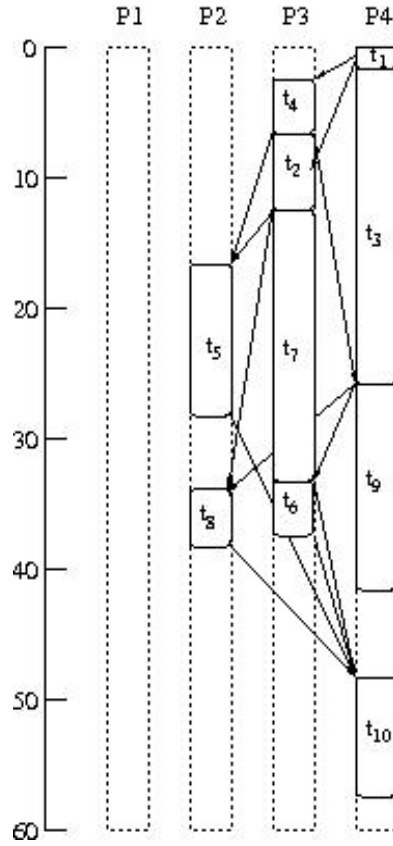


Figure IV.1 Sample Output Schedule of LHEFT algorithm for task graph given in Figure II-1 (resulting makespan is 57.16)

The HEFT algorithm uses an insertion-based policy. Inserting a task between previously scheduled tasks affects the execution time of latter task. Let task t_i and t_j scheduled on processor p and assume that t_i will be executed before t_j . The execution time of t_j will be determined by locality between these two tasks. In an insertion based scheduling policy, another task t_k may be scheduled between t_i and t_j , without violating precedence constraints. Now execution order in processor p becomes $t_i \rightarrow t_k \rightarrow t_j$, meaning t_j will execute after t_k and execution time of t_j will be determined by locality between t_j and t_k rather than t_i . In order to prevent any misleading information for tasks to be scheduled, algorithm must update schedule table considering this change in locality relation. Therefore, algorithm firstly updates tasks that are scheduled after the inserted task in the same processor queue; then,

children of these tasks will be updated. In this approach, algorithm only changes execution time of the task scheduled right after the inserted task (t_j). This will change *EFT* (maybe *EST* too) of t_j . Then, updates other related tasks *EFT* and *EST* values without altering execution times (since their locality relations haven't changed).

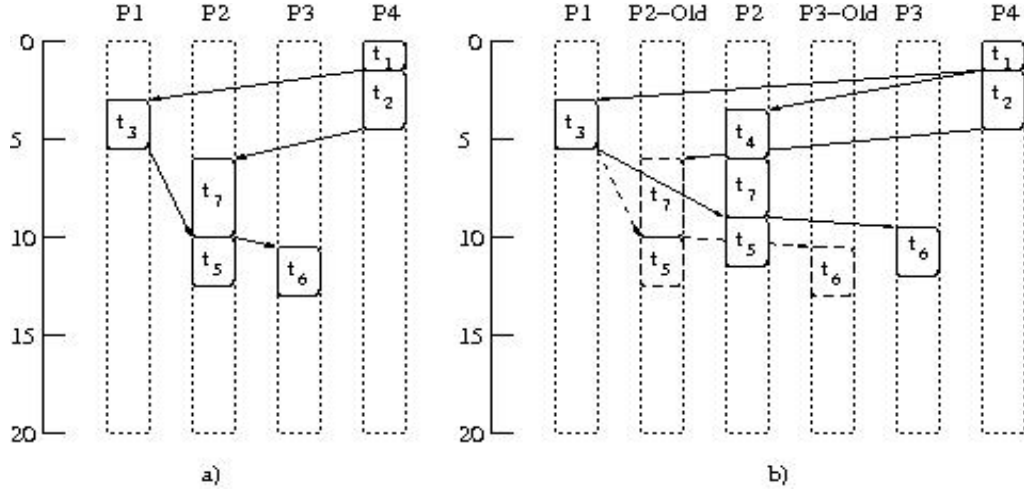


Figure IV.2 Example for Updates After Insertion in LHEFT

In Figure IV.2, (a) is the state of schedule right before inserting task t_4 . After the insertion of t_4 before t_7 , data reuse between tasks t_7 and t_4 changes *EFT* of the task t_7 . This change in the *EFT* creates a gap between tasks t_7 and t_5 , so LHEFT closes this gap by updating the *EST* and the *EFT* values of task t_5 . Since task t_6 is the only child of task t_7 , same update applied on task t_6 . In this update sequence, only t_7 's execution time has changed, updates on other tasks are only applied on their *EST* and *EFT* values.

IV.2. HYBRID GENETIC ALGORITHM FOR LOCALITY AWARE SCHEDULING

In this section we present the details of our GA-based solution

IV.2.1. Initial population generation

Ready queue is a list containing tasks whose parents are assigned, but not yet assigned to any processor. We start with an entry node and assign it to a random processor. After that we remove this node from ready queue and add children (if and

only if all their parents are assigned) to ready queue. Then, we select a random node from ready queue and repeat the same procedure until no task is left in ready queue. Algorithm creates an initial population using this random initialization. In order to improve performance, we make some modifications in population initialization to *seed* population with some improved individuals. This improvement is based on a simple list scheduling algorithm. Scheduling order of tasks is still created randomly, but assignment of these tasks into processors is done by using the heuristic which assigns tasks into processors giving best *EFT* value by following randomly created order of tasks.

IV.2.2. Genetic operators

GA has three fundamental operations to consider: *parent selection*, *crossover* and *mutation*.

IV.2.2.1. Parent Selection

For selection we considered tournament-based selection. Tournament size can be set to any number smaller than population size. Setting the size to one means the random selection of parents. In addition to tournament based selection, we implement random based monogamy selection. In this selection type, parents are selected randomly just like tournament based selection with size one, but any selected parent will not be re-selected as parent. In this selection, size of generated offspring population cannot exceed population size.

IV.2.2.2. Crossover

Nature of crossover in biology is to combine two parent chromosomes into a new chromosome. In our implementation, we have two types of crossover. First one is an order based crossover. Two chromosomes are split into two parts from a random crossover point. First half of the chromosome is copied into child. Then, second half is appended into child with order of appearance in second parent. Then, we swap first parent and second parent in order to generate a new child. An example of order-based crossover is given in Figure IV.3.

In the second crossover operator, called Coupling Crossover, task orders in the offspring are same as the order-based crossover given above. However, the mappings of tasks on processors are determined based on the strategy given in Figure IV.4.

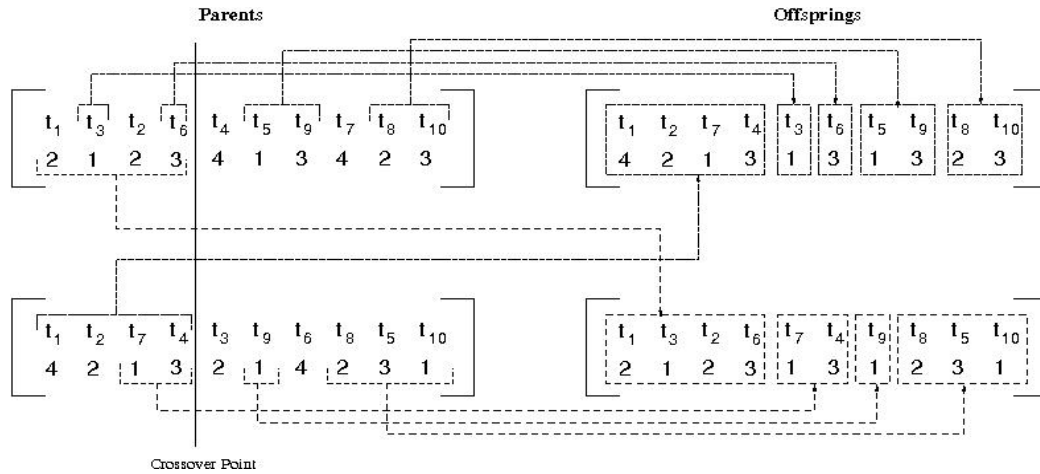


Figure IV.3 Applying order-based crossover operator

Tasks are assigned into processors using this *TaskPairs* list. Order of tasks in child is created by order based crossover, so index of each task is known. Algorithm tries to map tasks into processors beginning with the first element of *TaskPairs* list. Since it has more data reuse than others, it assigns tasks into processors of that pair.

1. *TaskPairs* : initially empty list for pairs
2. **for all** task t_i in pc_1
3. $t_{pair} \leftarrow$ next task in pc_1 assigned to the same processor with t_i
4. add t_i and t_{pair} as task pair (couple) to *TaskPairs* list
5. **end for**
6. repeat steps 2 to 5 for second parent pc_2
7. sort *TaskPairs* using data reuse between tasks in each task pair (couple)

Figure IV.4 Coupling Pseudo Code

In Figure IV.4 pc_1 and pc_2 denotes first and second parent to be used in crossover. For each task pair, which parent they came from and to which processor they are assigned in that parent are known. Tasks are assigned to their processor. This is repeated for each element in *TaskPairs* list. However, there are some conditions:

- each pair has a processor, so between indexes of tasks in pair there should be no other task assigned to the same processor;
- at least one task in pair should be unassigned (since both of them are assigned, there is no need to do any operation for that pair);
- if a task in pair is currently assigned, processor of that pair should be the same as assigned tasks' processor and rule 1 should be satisfied.

After looping through all pairs in *TaskPairs* there can be some tasks left unassigned. These tasks are assigned to the processor that provides maximum data reuse. When placing any unassigned task, algorithm checks if assigning task into that processor decreases locality. Considering this decrease (or maybe increase) task is assigned to the processor where it enhances maximum locality.

IV.2.2.3. Mutation

Three types of mutation operator [7] implemented in our study:

- M1: It randomly selects a task in chromosome and randomly changes its processor assignment. Figure IV.5.b mutates task t_9 by mapping it to processor p_1 .
- M2: It randomly selects a task and changes its scheduling order. In order to keep precedence constraints, the selected task is repositioned into an index from a range of minimum and maximum values where minimum is set with the index if its first scheduled child and maximum is set with the index of its last scheduled parent. Where task graph given in Figure II.1 and current chromosome given in Figure IV.5.c are considered, the minimum index of task t_9 is 6 and maximum index of task t_9 is 9. Therefore a node number is set from a range [6,9]; and t_9 is rescheduled into its new position in Figure IV.5.c.
- M3: This type compines first two types. (Figure IV.5.d)

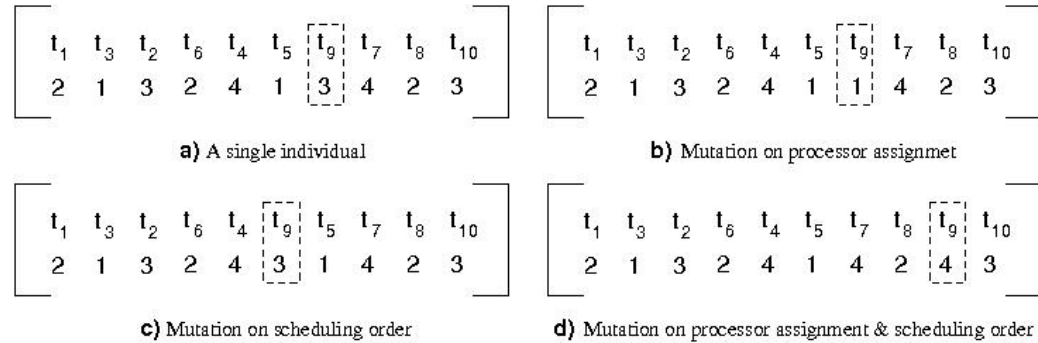


Figure IV.5 Illustrations of mutation operators

IV.2.3. Local Search

We considered two local search implementations.

1. The first one randomly picks a task from critical path tasks. Then, it tries to find a processor for selected task that gives highest data reuse.

2. The second method uses a list like *TaskPairs* list used in crossover, but pairs are global list of all tasks paired and ordered by data reuse amount. Local search starts searching this list and tries to put tasks on pair that has globally highest data reuse amount into the same processor. This process continues until a pair in global list is assigned to same processor.

Each local search implementation makes one try in each iteration because calculating fitness value has $O(v^2)$ in worst case, where v is the number of tasks (i.e. vertices in graph). This will increase the running time of GA, so we implement partial fitness calculator that calculates fitness value after a given position in chromosome. Since representation includes scheduling order, any change on a gene cannot affect *EFT* of previous tasks in scheduling order. Therefore, this partial calculator starts from changed gene and calculates change in fitness. This partial fitness calculator provides approximately 30% improvement, but still cannot remove limitation that comes from complexity.

IV.2.4. Survivor Selection

We implement three types of survivor selection mechanism.

- $(\mu + \lambda)$: This mechanism combines population with created offsprings. Then, it orders this combined space using fitness values and selects best of them to form new generation population.
- (μ, λ) : This mechanism selects individuals only from offspring population, so every generation has new individuals. This is an age based selection.
- *elitism*: This mechanism selects majority of new generation from offspring population, but keeps some percentage of best individuals from parent population.

If given parameter is between 0 and 1 then parameter is accepted as percentage and in each generation survivor count is decided by given percentage, if given parameter is bigger than one then in each generation survivor count is equal to given parameter.

IV.2.5. Termination Condition

We implement three types of termination condition:

- terminate at a predefined number of iterations
- terminate if fitness value of best solution does not improve during last k iterations

- terminate when average fitness of the population does not improve during last k iterations

Here k is a predefined number given in experimental study section. We will discuss which of these proposed types of GA elements to use in Experimental Study by showing statistics.

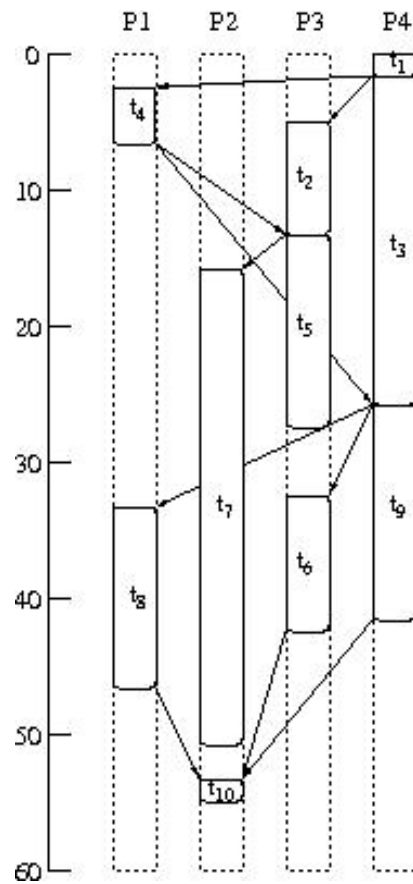


Figure IV.6 Sample Output Schedule for GA (resulting makespan is 54.20)

PART V

V. EXPERIMENTAL STUDY

This chapter presents the comparison study of the algorithms proposed in our study. We first present comparison metrics, which is followed by the details of the graph generator module. The experimental study is given in two parts: pre-experimentation for setting the GA parameters and a set of main experiments. It should be noted that all experiments are performed on a cluster of 11 machines, running Linux operating system. Each machine has a 2.4 GHz processor with 1 GB of memory.

V.1.PERFORMANCE METRICS

Comparisons of the algorithms are based on three metrics, which are normalized schedule length (NSL), frequency of best quality of output schedules, and running time.

- *Normalized Schedule Length (NSL)*: Main performance metric of a scheduling algorithm is the length of the schedule output, which is called *makespan*. Since algorithms are evaluated on a wide range of graph types, it is necessary to normalize the schedule length values. This normalized value is called the Normalized Schedule Length (NSL) [3], which is defined by Equation (V.1):

$$NSL = \frac{makespan}{\sum_{i_j \in CP_{MIN}} \min(w_{i,j})}, \quad (V.1)$$

In order to normalize makespan in lower bound, sum of the minimum execution times of tasks in critical path is used in the denominator of the equation. In heterogeneous task scheduling studies without locality knowledge this formula will always give lower bound value. But in our case because of locality any task's execution time may change depending on previous task executed in the same processor. Therefore, in some cases this lower bound value could be passed over. We can overcome this issue by subtracting maximum data reuse amount a task can have, from its execution time. But since we'd like to see impact of locality on makespan, we will keep this normalization value. This way, even task-data dependency changes, for the same sized graph, normalization value will be the same and we will see how locality affects makespan more clearly.

- *Frequency of Best Quality of Output Schedules*: Experimentation framework executes each algorithm for the same graph. Therefore, it can find which algorithm performs best on a particular graph. The number of occurrences of better quality schedules will demonstrate algorithms performance relative to the others.
- *Running Time*: The running time of an algorithm is the execution time passed for calculating the makespan. This metric shows the cost of each algorithm.

V.2.GENERATING RANDOM TASK GRAPHS

Graph generator module is implemented to compare our algorithms in various environments. Experimentation framework uses graph generator to create DAGs for our algorithms to be tested. This module aims to generate and set:

- Directed Acyclic Graph (DAG) of an application with communication costs.
- execution times of each task on each processor
- data load times of each task in the given DAG
- data re-use rates between tasks.

Our experimentation framework generates synthetic task graph using graph generator, then it executes each algorithm to generate output schedules. Those output schedules will be used in calculation of performance metrics and interpretation. Our graph generator is an extension of a previous work [3]. Our graph generator considers a set of parameters to determine the characteristics of task graphs, which are:

- Number of tasks in graph, (v).

- Number of processors, (ρ).
- Shape of the graph, (α).
- Out degree of nodes, (θ_{OUT}). Out degree denotes average connectivity of graph between each level, *i.e.* average number of dependencies to a task (communications sourced from a task).
- Task and machine heterogeneities, (ϕ_T and ϕ_M). ϕ_T is used for generating base computation cost of each task. Base computation cost show tasks' average computation costs on each machine, so this generated base value will be used for generating computation costs on each machine. To generate computation costs on each machine, ϕ_T is used in conjunction with ϕ_M . Base computation costs will be generated in range of $(0, 2\phi_T]$. Machine heterogeneity (ϕ_M) is the variation of computation costs of a task in different processors. Actual computation cost for each processor is computed by multiplying ϕ_T by ϕ_M .
- Communication to computation ratio, (CCR). It is the ratio of the average communication cost to the average computation cost. If a DAG's CCR value is very low, it can be considered as a computation intensive application and if CCR value is very high, it will be a communication intensive application.
- Task-data dependency rate, (δ_{data}). We discussed task-data dependency rate in Part 2. Task-data dependency rate denotes percentage of execution time spent for loading data. Computation cost is defined in equation (II.7). Data loading time is assumed to be the same on every processor. By definition of computation cost, data loading time should be included in generation of actual computation costs. Generator first generates base computation time using ϕ_T and this base value is assumed to include data loading time. Then, it generates a random percentage value using a Gaussian distribution with mean value as δ_{data} and multiplies this percentage value with base computation cost to generate data loading time. This data loading time is the same in every processor, so variation of actual computation cost is calculated using the difference of base calculation from data loading time. (*i.e.* $\phi_T - \phi_T \times \delta_{data}$).
- Reuse rate, (δ_{reuse}). Reuse rate is a relation between two tasks and coherent with task-data dependency (Equation (II.6)). It is generated using δ_{reuse} as the mean value in Gaussian distribution. Since reuse is a relation between two tasks, both

drr value (*i.e.* $drr_{i,j}$ and $drr_{j,i}$) should satisfy Equation (II.6). Therefore, the other relational drr value is calculated using generated value and Equation (II.6).

V.3.EXPERIMENTS FOR IDENTIFYING GA PARAMS

Ga-based framework implemented for this study has seven control parameters: crossover ratio, mutation ratio, crossover type, mutation type, local search, population size, number of generations.

There are two crossover types and three mutation types implemented. Early comparisons of crossover types show that Coupling-based crossover yields better performance than random-order based crossover depicted in Figure V.1.

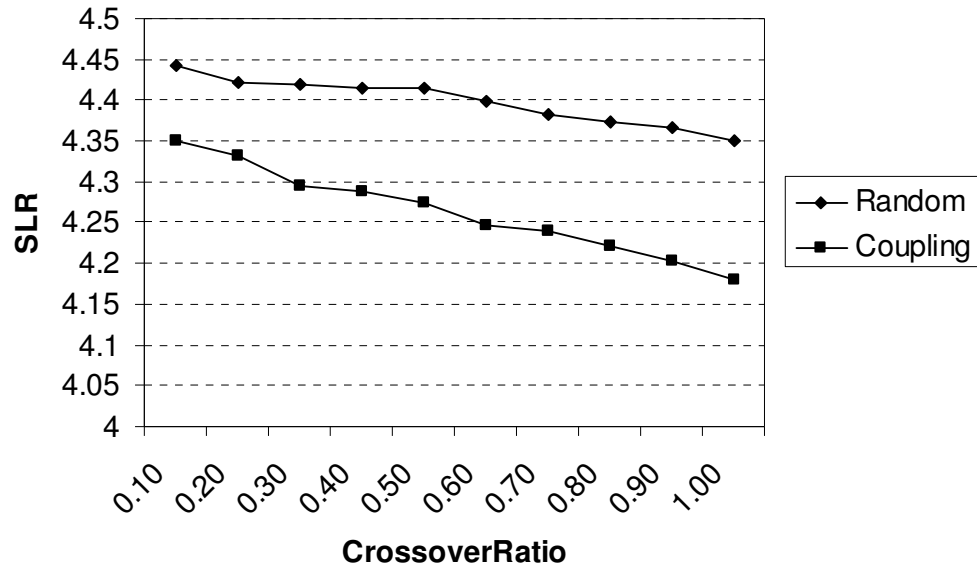


Figure V.1 Comparison of Crossover Types Without any Other Genetic Operator

However, in order to find suitable configuration we try to compare possible crossover and mutation rates for these six types. Since there are two different crossover types, firstly best fit mutation type for each of the crossover operator is found. Crossover rates vary from 0.1 to 1.0, incremented by 0.1 and mutation rates vary from 0.1 to 0.5 incremented by 0.05, which gives 90 different configuration for six crossover-mutation type pair. After running 90 different configuration for every crossover-mutation type pair we have six configurations, best of each. For these six configurations we will again run experimental tests in order to compare them. Each experimentation uses 1620 different graphs.

Table V-1 Comparison of Best Configuration for each Crossover and Mutation Type

Type Pair	Rates	NSL	Exec. Time	Impr.	Best Freq	Comb. Best
Random-Order	1.00-0.50	3.17	477.95	24.96	11	10.67%
Random-Assg.	1.00-0.50	2.38	482.43	39.05	962	87.48%
Random-Both	0.90-0.20	2.52	493.65	36.45	338	70.43%
Coupling-Order	1.00-0.50	3.01	2869.38	27.88	24	24.51%
Coupling-Assg.	0.20-0.50	2.54	865.90	36.35	210	65.81%
Coupling-Both	1.00-0.50	2.83	2584.34	31.50	85	41.27%

Results of comparisons for crossover and mutation types are shown in Table V-1. It is clear that “Random crossover” with “Assignment Mutation” is the best choice. It gives lowest in makespan, highest in fitness function improvement, and highest frequency of best makespan occurrence and highest percentage for combined comparison against all algorithms. For both types of crossover “mutating order” does not perform well. Cause of this is changing the order of tasks does not change its processor assignment. If the order of task changes, it makes difference if and only if its order in processor queue also changes. Since a task's order changes between its predecessors and successors, possibility of a change in the order of processor queue decreases. Random-order based crossover does not consider any problem specific knowledge, but it is clear that it gives good quality schedules with random mutations. On the other hand, “coupling” based crossover also does not yield an acceptable performance in contrast to Figure V.1; since coupling based crossover is highly dependent on processor assignments. Any change in these assignments can degrade performance very badly. It can be seen from the results that mutation operators do not improve makespan as much as they improve random-order based crossover.

Local search is applied after every genetic operator. We run experimentation for our two implementations. Local search for *critical path improvement*, which selects a random task in critical path and tries to decrease the *EFT* value of that task, gives better result on 67 per cent of experiments for the best configuration discussed previously. On overall (Table V-2) local search for critical path improvement gives better results.

Table V-2 Comparison of Local Search Methods for Crossover and Mutation Types

Crossover	Mutation	LS-CP	LS-Loc
Random	Assg.	67%	33%
	Both	74%	26%
Coupling	Assg.	68%	32%
	Both	70%	30%

Task graphs with higher number of tasks increases search space; therefore, population size should be increased with an increase in task size. Number of generations in GA also depend on problem size. For a small sized problem, convergence may occur very fast but for a bigger problem space convergence might not be that fast. We define population size by $P = k_1 \times v$ and number of generations by $I = k_2 \times v$. Here, k_1 and k_2 coefficients varied in set $S = \{1,2,3,5\}$. If the set S enlarged by adding larger values, GA suffers from the longer execution times.

Table V-3 Comparison study of Population Coefficients

k_1	NSL	Exec. Time (ms)	Best Freq	Comb. Best
1	12.27	335.21	38	132.00
2	12.11	682.64	130	372.67
3	12.01	1054.39	229	506.00
5	11.90	1878.33	500	666.00

Table V-4 Comparison study of Iteration Coefficients

k_2	NSL	Exec. Time (ms)	Best Freq	Comb. Best
1	12.24	362.41	80	169.00
2	12.10	695.33	197	434.33
3	12.02	1022.83	299	575.33
5	11.96	1707.27	555	666.00

Table V-3 and

Table V-4 shows that larger population size and longer iterations yields best results. But as the coefficients grow larger, execution times becomes longer. Therefore, we found it is feasible to set k_1 and k_2 with value of 3.

V.4.COMPARISON STUDY BASED ON RANDOMLY GENERATED TASK GRAPHS

We use parameters for GA as discussed in previous chapter. Half of the initial population is generated by seeding them with a list scheduling algorithm as described in section IV.2.1 and the other half is generated by randomly.

Task graphs for experimentation is classified into two different heterogeneity types with respect to the values of the task and machine heterogeneity ranges, which are the low task-low machine heterogeneity (LoLo) type and the high task-high machine heterogeneity (HiHi) type. Task heterogeneity range is 60 for low value and 600 for

high value; and machine heterogeneity range is set to 2 for low value and 10 for high value. Due to some similarities, low task-high machine heterogeneity and high task-low machine heterogeneity types are ignored.

V.4.1. Comparison with respect to Graph Size

First experiment is based on effect of the size of the graph. Graph size determined by number of tasks that exists in graph. For this experiment graph size is varied in set $G_{SIZE} = \{20,40,60,80,100\}$. For each experiment; CCR values are set to 1.0 and fan-out degree is set to 4. For each element in set G_{SIZE} 810 experiments are executed. In Tables Table V-5 and Table V-6, values of NSL and running time are the average values of the 810 different graphs.

Table V-5 Comparisons of Algorithms with Respect to Graph Sizes for LoLo Heterogeneity Type

Algorithm	Metric	Graph Size				
		20	40	60	80	100
HGA	NSL	2.45	2.60	2.70	2.76	2.77
	Time (ms)	32.72	252.22	836.22	1993.78	3848.56
	Freq. Of Best Makespan	439	542	590	622	624
LHEFT	NSL	2.53	2.70	2.82	2.89	2.89
	Time (ms)	0.00	0.23	0.81	0.46	0.42
	Freq. Of Best Makespan	133	107	102	82	106
HEFT	NSL	2.59	2.77	2.88	2.95	2.96
	Time (ms)	0.00	0.12	0.15	0.54	0.11
	Freq. Of Best Makespan	81	44	40	36	46
DLS	NSL	3.46	3.73	3.91	4.03	4.03
	Time (ms)	0.00	0.73	2.10	3.20	3.75
	Freq. Of Best Makespan	6	1	0	0	0
LS	NSL	4.06	5.02	5.90	6.88	7.28
	Time (ms)	0.00	0.00	0.05	0.21	0.09
	Freq. Of Best Makespan	0	0	0	0	0

When Table V-5 is examined, HGA based scheduling algorithm has a very good average NSL value and it gives best results among five algorithms. The HEFT algorithm has performed very well and proven to be a strong algorithm on heterogeneous environments; even in this specialized case of locality it gives good results. Therefore, the LHEFT algorithm gives much better results than the HEFT algorithm. But as size of the graph becomes large, the HEFT algorithm loses ground to HGA. Only the LHEFT algorithm seems to keep its performance. DLS and LS algorithms does not perform well on this experimentation.

Table V-6 Comparisons of Algorithms with Respect to Graph Sizes for HiHi Heterogeneity Type

Algorithm	Metric	Graph Size				
		20	40	60	80	100
HGA	NSL	1.91	2.00	2.03	2.08	2.10
	Time (ms)	240.09	409.47	602.33	798.27	993.42
	Freq. Of Best Makespan	763	762	748	726	712
LHEFT	NSL	2.05	2.13	2.15	2.20	2.20
	Time (ms)	0.00	0.28	0.78	0.70	0.64
	Freq. Of Best Makespan	38	31	35	51	59
HEFT	NSL	2.07	2.15	2.18	2.22	2.23
	Time (ms)	0.01	0.14	0.26	0.68	0.23
	Freq. Of Best Makespan	23	17	27	29	29
DLS	NSL	5.49	5.76	5.82	5.96	5.87
	Time (ms)	0.00	0.86	1.86	2.98	3.99
	Freq. Of Best Makespan	0.00	0.00	0.00	0.00	0.00
LS	NSL	5.64	6.53	7.49	8.71	9.33
	Time (ms)	0.00	0.02	0.16	0.30	0.07
	Freq. Of Best Makespan	0.00	0.00	0.00	0.00	0.00

As expected, HGA algorithm has a very good performance but lacks running time. Even at small sized graphs, GA works very slow. As its complexity shows, the DLS algorithm works longer than other algorithms. The LHEFT algorithm has an increase in running time relative to similar HEFT algorithm, this is because of modified version insertion based policy. The LHEFT algorithm has to update schedule table if an insertion occurs, because locality relation changes in schedule affects whole algorithm.

The results given in Table V-6 is similar to the results given in Table V-5 for DLS and LS algorithms. HGA has outperformed all other algorithms. Especially if frequency of best schedule occurrences is inspected it can be seen GA has 763 best schedules out of 810 experiments (94%). Although it gives an outstanding performance, it still lack slow running time.

V.4.2. Comparison with respect to CCR

In this experiment algorithms are compared with the effects changes in CCR values. CCR is the communication to computation ratio. If CCR has a low value, it means that application is computation intensive with low communication costs. This ratio does not affect number of connections rather it affects weights of communications on execution of application. If CCR has a high value, then it means that application is

communication intensive. It takes long times to communicate from one processing element to an other.

Graph size is set to 60 graphs and fan-out degree is set to 4 in the experiments . CCR value is varied in set $G_{CCR} = \{0.1, 0.5, 1.0, 5.0, 10.0\}$. For each element in set G_{CCR} , 810 experiments run. In Table V-7 and Table V-8, values of NSL and running time are average values of the 810 different graphs.

Table V-7 Comparisons of Algorithms with Respect to CCR values for LoLo Heterogeneity Type

Algorithm	Metric	CCR Value					
		0.1	0.5	1	2	5	10
HGA	NSL	0.98	1.67	2.76	4.93	12.36	23.91
	Time (ms)	832.60	832.80	833.12	834.89	823.05	814.05
	Freq. Of Best Makespan	791	774	582	92	0	0
LHEFT	NSL	1.10	1.81	2.87	4.76	9.81	15.08
	Time (ms)	0.75	0.60	0.59	0.59	0.56	0.72
	Freq. Of Best Makespan	16	22	121	267	304	314
HEFT	NSL	1.17	1.86	2.95	4.86	10.09	15.68
	Time (ms)	0.19	0.14	0.19	0.12	0.20	0.17
	Freq. Of Best Makespan	2	8	37	110	126	145
DLS	NSL	2.26	2.96	3.98	5.81	11.23	17.93
	Time (ms)	1.99	2.25	2.11	2.02	2.36	2.06
	Freq. Of Best Makespan	0	0	0	6	26	43
LS	NSL	3.53	4.57	5.98	8.41	16.89	29.38
	Time (ms)	0.10	0.15	0.09	0.12	0.06	0.05
	Freq. Of Best Makespan	0	0	0	0	0	0

When Table V-7 is examined, HGA based scheduling algorithm outperforms all other algorithms by giving 791 best schedules among 810 graphs (97,7%). It also gives 11% better NSL average than the LHEFT algorithm. As the CCR value grows and communication dominates execution time, although HGA keeps significant performance, other algorithms began to perform better. The most significant one is the LHEFT algorithm. Its inherited aspects (especially upward rank definition) from the HEFT algorithm helps to gain the edge in communication intensive environments. The HEFT algorithm, even though it has no information about locality, shows a significant improvement as CCR value grows. DLS and LS algorithms does not perform well on this experimentation.

Table V-8 Comparisons of Algorithms with Respect to CCR values for HiHi Heterogeneity Type

Algorithm	Metric	CCR Value					
		0.1	0.5	1	2	5	10
HGA	NSL	1.03	1.42	2.06	3.49	8.37	15.9
	Time (ms)	595.32	595.73	597.36	597.20	592.56	588.49
	Freq. Of Best Makespan	784	776	739	574	159	27
LHEFT	NSL	1.12	1.52	2.18	3.62	8.09	13.93
	Time (ms)	0.69	0.83	0.75	0.95	0.70	0.84
	Freq. Of Best Makespan	18	25	47	97	238	277
HEFT	NSL	1.15	1.55	2.21	3.65	8.15	14.08
	Time (ms)	0.12	0.14	0.22	0.28	0.12	0.25
	Freq. Of Best Makespan	14	15	18	59	176	186
DLS	NSL	4.60	5.17	5.83	7.33	11.48	17.20
	Time (ms)	1.86	2.19	1.81	1.84	1.98	1.96
	Freq. Of Best Makespan	0	0	0	0	2	11
LS	NSL	5.79	6.74	7.44	9.30	15.90	24.38
	Time (ms)	0.02	0.09	0.10	0.12	0.20	0.15
	Freq. Of Best Makespan	0	0	0	0	0	0

For the HiHi heterogeneity type (Table V-8) there is not much difference in average NSL values; HGA still performs better than the others. As same in the LoLo case, increase in CCR value decreases performance of HGA but not as quick as in LoLo case. When CCR is 2.0, the HGA keeps a significant performance.

On both Graph size based and CCR based experimentations HGA seems to perform better on HiHi heterogeneity case. In HiHi heterogeneity cases both task heterogeneity and machine heterogeneity varies in large intervals. This partially decreases effects of communications and increase the importance of locality. For example if a task executes 400 seconds in some processor, it can execute 4000 seconds in an other when HiHi heterogeneity case is considered. If this task assumed to load data for 250 seconds, then this means locality improvement in fastest machine decreases execution time much more than slower machine. Since HGA exploits locality in local search operator it keeps its performance in high heterogeneity cases. The LHEFT algorithm also combines its legacy from the HEFT algorithm and locality issues so that it keeps performance in various experiments. The reason behind the DLS algorithm's low performance can be in dynamic level definition. Since dynamic level definition considers execution start time rather than execution finish time on a particular processor, it may not be able to exploit locality. The LS algorithm is a greedy algorithm performing on processors rather than nodes. This causes

schedule to span onto nearly all processors, even if its not necessary, thus increasing communications and overall makespan.

V.4.3. Comparison with respect to Locality

The impact of locality can be seen through some more details. Figure V.2, Figure V.3, and Figure V.4 shows the graphs about the impact of task-data dependency and data reuse on generated makespan. For the figures, graph size is set to 60 and fan-out degree is set to 4. For three different situations, three different plot drawn with CCR values 0.1, 1.0, 10.0. Inspected task-data dependency rates are chosen from the set $G_{DATA} = \{0.2, 0.5, 0.8\}$ and data reuse rates are chosen from the set $G_{DRR} = \{0.1, 0.5, 0.9\}$. Small task-data dependency rates means that tasks are more computation dominated, and high rates means tasks require significant amount of data for execution. Similarly small data reuse rates means task share small amounts of data and vice versa.

Looking at figure Figure V.2, for computation intensive task graphs as the task-data dependency increases, more significant improvement in NSL is observed. Data reuse effect is more significant in high rates of task-data dependency. When task-data dependency is 0.2, the improvement with the increasing data reuse is between 10% and 20%; on the other hand when task-data dependency is 0.8, the improvement with the increasing data reuse is between 35% and 50%.

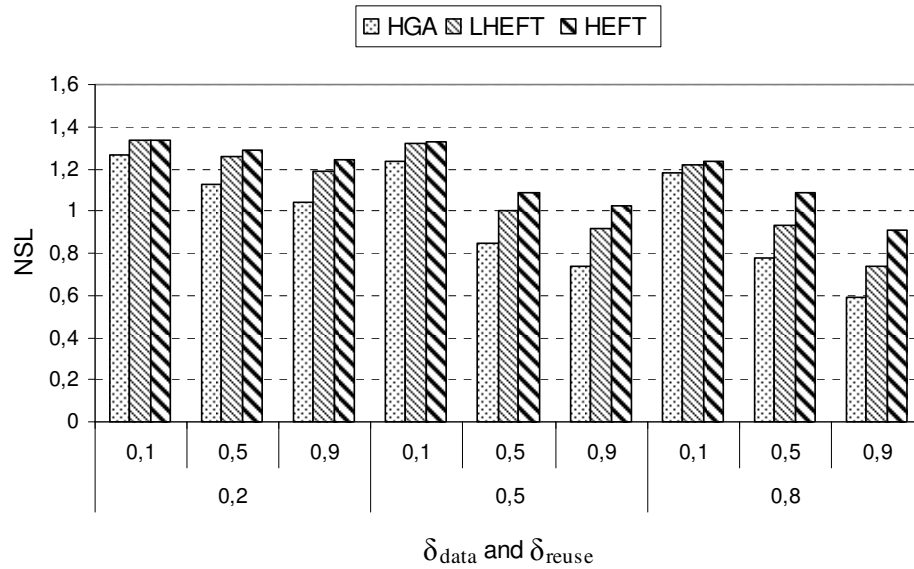


Figure V.2 Effect of Data Amount and Data Reuse with CCR=0.1

As the HEFT algorithm does not have any knowledge of locality, its results can be used as a reference point for the other algorithms. When the task-data dependency or data reuse are at low rates, impact of locality seems to fade. Definition of the problem becomes very similar to heterogeneous tasks scheduling problems without locality. The HEFT algorithm seems to perform well at those situations. But as the impact of locality increases the HEFT algorithms performance decreases, but the other algorithms with the knowledge of locality outperforms the HEFT algorithm. On the other hand, our extension of the HEFT algorithm outperforms all other algorithms with the exception of HGA. By its definition the LHEFT algorithm uses the HEFT algorithm as a base and extends it with the knowledge of locality. So in all our cases the LHEFT algorithm gives better result than the HEFT algorithm. And at the low rates of locality aspects it performs very similar to the HEFT algorithm.

Looking at the Figure V.3, same comments are valid. But, with the CCR value increased, graphs are no longer computation intensive and the effect of locality is not as clear as CCR=0.1.

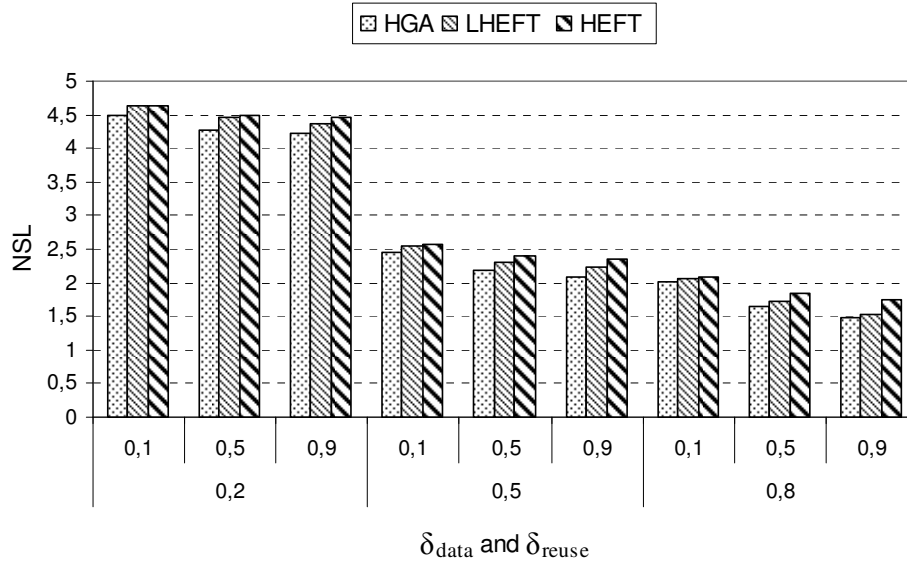


Figure V.3 Effect of Data Amount and Data Reuse with CCR=1.0

In the Figure V.4, CCR is set to 10.0 and graphs are now dominated by communication costs. Increase in task-data dependency still improves NSL but impact of data-reuse is not as strong as in lower CCR values. Also HGA suffers from the increased CCR. HGA's operators are implemented with the locality and computation costs in mind. Therefore, it cannot perform well on higher CCR values.

LHEFT algorithm seems to keep its performance on both high and low values of CCR.

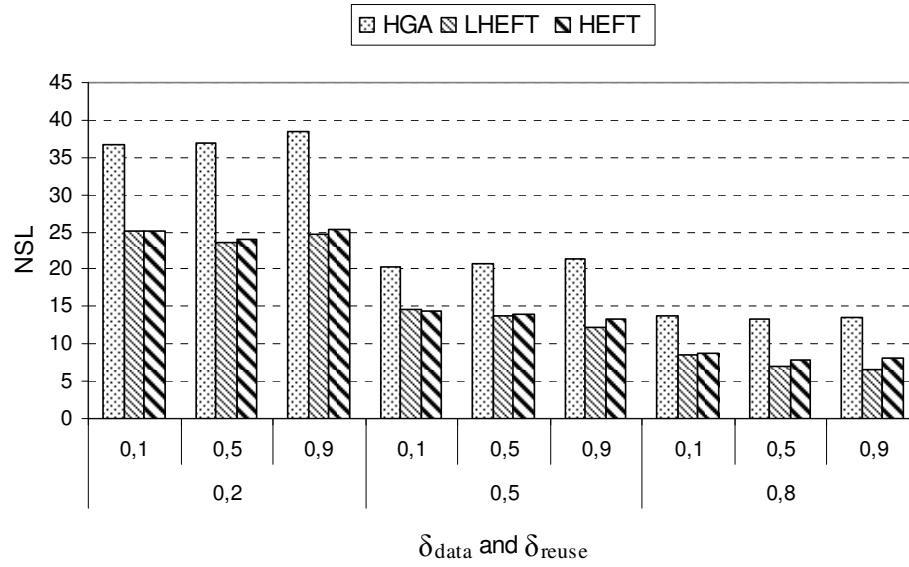


Figure V.4 Effect of Data Amount and Data Reuse with CCR=10.0

PART VI

VI. CONCLUSION

Locality based compiler optimization techniques are well studied and documented topic in literature. These studies mainly focus on loop and data-layout optimizations. Some recent studies focus on multiprocessor system on a chip (MPSoCs) in OS level. Both these studies inspect code of the application first in compile-time and either optimize it using loop transformations in compile-time or use data-layout optimizations in running time in order to enhance locality. Only a few studies try to utilize memory access relations between tasks.

As far as we know, there is no study in the literature that considers locality concept in heterogeneous computing environments. We define the locality in terms of task-data dependency and data reuse rates.

The main motivation of this thesis is to study the impact of data locality on scheduling in heterogeneous computing environments. In this study, we first present a new model for locality-aware task scheduling. Then we propose two task scheduling algorithms based on locality concept, which are Locality-aware Heterogeneous Earliest Finish Time (LHEFT) and a hybrid genetic algorithm. The proposed algorithms are compared with the leading algorithms for heterogeneous computing environments presented in the literature, which are Heterogeneous Earliest Finish Time (HEFT), Dynamic Level Scheduling (DLS) and Locality-Aware Scheduling (LS).

Graph generator is developed as an extension of a previous study [3]. Extended graph generator is responsible for generating data fetching times as task-data dependency and data reuse rates in addition to synthetic task graphs. On the

experiments we see that exploiting locality yield good performance if computation cost are bottlenecks of the environment. When the performance of the algorithms are examined, hybrid GA outperforms the others in many occasions, especially in computationally intensive environments. On the other hand it has a major downside: high execution time. GA's, by their nature, has a high running time. Some modifications like decreasing iterations or population sizes may improve running time by sacrificing quality on output schedules. The LHEFT algorithm is a very strong alternative. With the legacy of the HEFT algorithm, it shows very good performance on all occasions, even where GA fails. Of the referenced algorithms only LS considers locality. It does not perform well in heterogeneous environments since it is designed for OS-level scheduling.

REFERENCES

- [1] M. R. Garey and D. S. Johnson. "Computers and Intractability: a guide to the theory of NP-completeness" . W. H. Freeman, **(1979)**.
- [2] J. D. Ullman. "Np-complete scheduling problems." Journal of Computer and System Sciences, vol. 10, pp. 384-393, **(1975)**.
- [3] Topcuoglu, H., Hariri, S., and Wu, M. "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing". IEEE Trans. Parallel Distrib. Syst. 13, 3, 260-274, **(2002)**.
- [4] Sih, G. C. and Lee, E. A.. "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures." IEEE Trans. Parallel Distrib. Syst. 4, 2, 175-187, **(1993)**.
- [5] El-Rewini, H. and Lewis, T. G. "Scheduling parallel program tasks onto arbitrary target machines." J. Parallel Distrib. Comput. 9, 2, 138-153, **(1990)**.
- [6] P. Shroff, D.W. Watson, N.S. Flann, and K. Freund, "Genetic Simulated Annealing for Scheduling Data-Dependent Tasks in Heterogeneous Environments," Proc. Heterogeneous Computing Workshop, pp. 98-104, **(1996)**.
- [7] Betül Demiroz, Haluk Rahmi Topcuoglu: "Static Task Scheduling with a Unified Objective on Time and Resource Domains". Comput. J. 49(6): 731-743 **(2006)**
- [8] Claudia Leopold. "Generating Structured Program Instances with a High Degree of Locality", Proceedings of 8th Euromicro Workshop on Parallel and Distributed Processing, pp. 267-274 **(2000)**
- [9] Eiben, A.E. and Smith, J.E. "Introduction to Evolutionary Computing", Springer-Verlag, Berlin **(2003)**

- [10] Wolpert, D.H., Macready, W.G., "No Free Lunch Theorems for Search, Technical Report" SFI-TR-95-02-010 (Santa Fe Institute), **(1995)**.
- [11] Wolpert, D.H., Macready, W.G., "No Free Lunch Theorems for Optimization" IEEE Transactions on Evolutionary Computation 1, 67, **(1997)**.
- [12] Kandemir, M. and Chen, G., "Locality-Aware Process Scheduling for Embedded MPSoCs". In Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2 **(2005)**.
- [13] M. Iverson, F. Ozguner and G. Follen, "Parallelizing Existing Applications in a Distributed Heterogeneous Environment" Proc. Heterogeneous Computing Workshop, pp. 93-100, **(1995)**
- [14] Dhodi, M. K., Ahmad, I. and Storer R. "SHEMUS: synthesis of heterogenous multiprocessor systems". Microprocess. Microsys., 19, 311-319 **(1995)**
- [15] Imtiaz Ahmad, Muhammad K. Dhodhi: "Multiprocessor Scheduling in a Genetic Paradigm". Parallel Computing 22(3): 395-406 **(1996)**
- [16] Lee Wang, Howard Jay Siegel, Vwani P. Roychowdhury, Anthony A. Maciejewski: "Task Matching and Scheduling in Heterogenous Computing Environments Using a Genetic-Algorithm-Based Approach". J. Parallel Distrib. Comput. 47(1): 8-22 **(1997)**
- [17] Ricardo C. Corrêa, Afonso Ferreira, Pascal Rebreyend: "Scheduling Multiprocessor Tasks with Genetic Algorithms". IEEE Trans. Parallel Distrib. Syst. 10(8): 825-837 **(1999)**
- [18] Frode Eika Sandnes, Graham M. Megson: "Improved Static Multiprocessor Scheduling using Cyclic Task Graph"s: A Genetic Approach. PARCO 1997: 703-710, **(1997)**
- [19] P. Rebreyend, F.E. Sandnes, "G.M. Megson: Static Multiprocessor Task Graph Scheduling in the Genetic Paradigm: A Comparison of Genotype Representations". Ecole Normale Supérieure de Lyon, France. Research Report 98-25 **(1998)**
- [20] Page, A. J. and Naughton, T. J. "Framework for Task Scheduling in Heterogeneous Distributed Computing Using Genetic Algorithms". Artif. Intell. Rev. 24, 3-4, 415-429, **(2005)**

- [21] Holland, John H, "Adaptation in Natural and Artificial Systems", University of Michigan Press, Ann Arbor, **(1975)**.
- [22] Dimitriou, Georgios. "Simulation of Static and Dynamic Task Scheduling on Multiprocessor Systems". M.S. Thesis, University of Illinois at Urbana Champaign, **(1994)**
- [23] Guilin Chen, Guangyu Chen, Ozcan Ozturk, Mahmut T. Kandemir: An Adaptive Locality-Conscious Process Scheduler for Embedded Systems. IEEE Real-Time and Embedded Technology and Applications Symposium 2005: 354-364. **(2005)**
- [24] Guilin Chen, Guangyu Chen, Ozcan Ozturk, Mahmut T. Kandemir: "Exploiting Inter-Processor Data Sharing for Improving Behavior of Multi-Processor SoCs." ISVLSI 2005: 90-95. **(2005)**
- [25] Tilman Wolf, Prashanth Pappu, Mark A. Franklin: "Predictive scheduling of network processors. Computer Networks" 41(5): 601-621 **(2003)**
- [26] T.Yang and A.Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors", IEEE Trans. Parallel and Distributed Systems, vol. 5, no.9, pp. 951-967, **(1994)**
- [27] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," Proc. Int'l Conf. Parallel Processing, vol. 2, pp. 1-8,**(1988)**
- [28] Jing-Chiou Liou , Michael A. Palis, A Comparison of General Approaches to Multiprocessor Scheduling, Proceedings of the 11th International Symposium on Parallel Processing, p.152-156, April 01-05, **(1997)**
- [29] J. Liou and M.A. Falls, "An Efficient Clustering Heuristic for Scheduling DAGs on Multiprocessors," Proc. Symp. Parallel and Distributed Processing, **(1996)**.
- [30] Boontee Kruatrachue , Ted Lewis, Grain Size Determination for Parallel Processing, IEEE Software, v.5 n.1, p.23-32, **(1988)**
- [31] Y.-C. Chung , S. Ranka, Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors, Proceedings of the 1992 ACM/IEEE conference on Supercomputing, p.512-521, November 16-20, 1992, Minneapolis, Minnesota, United States **(1992)**

- [32]** I. Ahmad and Y. Kwok "A New Approach to Scheduling Parallel Programs Using Task Duplication," Proc. Int'l Conf Parallel Processing, vol. 2, pp. 47-51, **(1994)**
- [33]** M. Y. Wu , D. D. Gajski, Hypertool: A Programming Aid for Message-Passing Systems, IEEE Transactions on Parallel and Distributed Systems, v.1 n.3, p.330-343, **(1990)**

RESUME

Alper KÖSE

PHONE : +90 533 410 53 89
BIRTH : April 1, 1982 / BURSA
MARITAL STATUS : Single
E-MAIL : alperkose@gmail.com

Education

MS : Marmara University, Istanbul, Turkey
Computer Engineering
2004-present
BS : Marmara University, Istanbul, Turkey
Computer Science Engineering
2000 – 2004

Work Experience

04/2005 – Present : Software Engineer,
Siemens A.Ş.,
İstanbul, Turkey
07/2004 – 3/2005 : Software Engineer,
STEFF Technology A.Ş.,
İstanbul, Turkey.