Adaptive Scheduling Using Support Vector Machine on Heterogeneous Distributed Systems

by

Yongwon Park

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 6, 2011

Keywords: Heterogeneous Computing, Task mapping, Support Vector Machine

Abstract

Since the advent of the modern von Neumann computer in the 1940s, startling advances have been made in computing technology with the creation of innovative, reliable, and faster electronic components, from vacuum tubes to transistors. Computing power has risen exponentially over relatively brief periods of time as Moore's law projected a salient trend for the growth in memory-chip performance, estimating the capacity of the integrated circuit to double every 18 months. Although these developments were essential in solving computationally intensive problems, faster devices were not the sole contributing factor to high performance computing. Since electronic processing speeds began to approach limitations imposed by the laws of physics, it became evident that the performance of uniprocessor computers was limited. This has led to the prominent rise of parallel and distributed computing. Such systems could be *homogeneous* or *heterogeneous*. In the past decade homogeneous computing has solved large computationally intensive problem by harnessing a multitude of computers via a high-speed network. However, as the amount of homogeneous parallelism in applications decreases, the homogeneous system cannot offer the desired speedups. Therefore, heterogeneous architectures to exploit the heterogeneity in computations came to be a critical research issue. In heterogeneous computing (HC) systems consisting of a multitude of autonomous computers, a mechanism that can harness these computing resources efficiently is essential to maximize system performance. This is especially true in mapping tasks to heterogeneous computers according to the task computation type, so as to maximize the benefits from the heterogeneous architecture.

The general problem of mapping tasks onto machines is known to be NP-complete, as such, many good heuristics have been developed. However, the performance of most heuristics is susceptible to the dynamic environment, and affected by various system variables.

Such susceptibility makes it difficult to choose an appropriate heuristic. Furthermore, an adaptable scheduler has been elusive to researchers. In this research, we show that using a support vector machine (SVM) an elegant scheduler can be constructed which is capable of making heuristic selections dynamically and which adapts to the environment as well. To the best of our knowledge, this research is the first use of SVM to perform schedule selections in distributed computing. We call the novel meta-scheduler, support vector scheduler (SVS). Once trained, SVS can perform the schedule selections in $O(n)$ complexity, where n is the number of tasks. Using rigorous simulations, we evaluated the effectiveness of SVS in making the best heuristic selection. We find that the average improvement of SVS over random selection is 29%, and over worst selection is 49%. Indeed, SVS is only 5% worse than the theoretical best selection. Since SVS contains a structural generalization of the system, the heuristic selections are adaptive to the dynamic environment in terms of task heterogeneity and machine heterogeneity. Furthermore, our simulations show that the SVS is highly scalable with number of tasks as well as number of machines.

Acknowledgments

Looking back since I studied in America, I am surprised and at the same time very grateful for all I have achieved throughout the study. It has certainly shaped me as a Computer Scientist and has led me where I am now.

I would like to first express my deepest gratitude to my adviser, Dr. Sanjeev Baskiyar, for his excellent guidance and his invaluable comments from the beginning of this study to the completion. His valuable comments about the need for this dissertation helped me to keep going when times were tough. He inspired me and encouraged me to a deeper understanding of research work before I thought I could do any research at all. He has enlightened me through his wide knowledge of scientific work and his deep insights where it should go and what is necessary to get there. I would like to thank my dissertation committee members, Dr. Cheryl Seals and Dr. Dean Hendrix, for many suggestions that have improved this dissertation. I also wish to acknowledge Dr. Adit Singh, who provided interesting discussions and reviews for the dissertation. Thanks to Dr. Caio Soares for the careful reading of early draft of this document. I would like to thank Dr. Umphress for his many years of service for my assistantship in the doctoral program. I would like to thank the graduate school and the Department of Computer Science and Software Engineering staff members for assisting me with administrative tasks for completing my doctoral program. A very special thanks goes to Dr. Kai Chang and Dr. Homer Carlisle for their support with my academic career. I thank my cousin, Taewon Kim, who is currently pursuing postdoctoral research at Stanford University, for his help with preparing study in America. Many thanks to those friends in Auburn who shared friendship long years with me playing sports together.

Finally I wish to express my special gratitude to my parents for their emotional support. I can't possibly thank my parents enough. They were always supporting me and encouraging me with their best wishes.

I finish with a final silence of gratitude for my life.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the advent of the modern von Neumann computer in the 1940s, startling advancements have been made in computing technology with the availability of innovative, reliable, and fast electronic components, from vacuum tubes to transistors. The clock speed of early computers in the millisecond level increased up to a factor of a million in fifty years; accordingly, uniprocessors' speed increased by a factor of 10 every seven years. The technical advancement achieved in the development of memory-chip performance, roughly followed Moore's law, which projected the capacity of the integrated circuit double every 18 months. Concurrent with these developments, there has been increasing demand for computing resources. Since computers make it possible to perform scientific computations in far less time than would otherwise be possible, its use extended greatly to various areas, and the dependence on computers increased rapidly. High performance computing has a history that has been developed within crisis management, placing demand on high-performance computations for modeling natural phenomenon relevant to crises, such as severe storms, earthquakes, and atmospheric dispersion of toxic substances [110]. Similarly, there have been various efforts for improving the performance of computers to meet diverse needs for computing resources, giving rise to great interests in high-performance computing. In one such effort, the desire to speed up computation led to the development of a variety types of supercomputers. They were usually devised for highly calculation-intensive tasks, *grand challenge* problems such as weather forecasting, climate research, biological macromolecules, and so forth. When speculating on the evolution of supercomputers, we can find a certain trend in speed-gains of computers. In early supercomputers, the speed-up was achieved by the development of fast scalar processors, serial processors that operated in the simplest operation mode. The next powerful form of multiprocessing was provided by vector processors, which are particularly good at performing vector and matrix operations, and further by parallelizing computations through the connection of a massive collection of processing units. The computer architecture based on a uni-processor in which its performance mainly depends on its clock speed has been changed to the form of multi-processor. Also, as the commodity development of computers with powerful micro-processor is available, relatively inexpensive computers such as personal computers and workstations came to be exploited in high-performance computing by combining these computing resources in a cooperative way using software technique to create a computing power required for computation intensive tasks, e.g., Beowulf clustering system [101] was able to create a great computing power needed by connecting personal computers via widely available networking technology running any one of several open-source operating system such as Linux. Nowadays, high-performance computing seems to continue to favor such an integration of existing computing resources, influenced by the fact that electronic processing speeds began to approach limitations imposed by the laws of physics, rather than develop a new fast processor. Furthermore, the advent of high-speed networks such as Fiber Distributed Data Interface made the construction of large scale of parallel and distributed (PDC) systems with the least communication overhead possible. Thus, even computers placed a physically long distance apart can be seamlessly connected with the help of data communication and system operating technology. Therefore, it is expected in the future that the main method of obtaining computing power needed for computation demanding tasks will be an integrated collection of computing resources using high-speed network. In this sense, PDC is currently considered a promising technique that can provide a vast amount of computing power in a cost-effective way, and is able to provide an underlying environment on which high-profile technologies such as pervasive and nomadic computing [99] can flourish. For example, millions of computers can be harnessed cooperatively with the dedication of part of their computing resources via the Internet. The collected computing power can be an amount so formidable that it would surpass even the fastest supercomputers. With this gradual change in the computing paradigm, it is tempting to think that PDC is such a magical technology that it can solve computational demanding problems by connecting a multitude of computers via a high-speed network, and further breach barriers which would be posed by the physical limitations of single processor. However, the optimistic vision may not be achievable by just connecting a multitude of computers; in fact, beyond this vision we are faced with many challenging problems to be solved in order to make it a reality

PDC has been developed in the form of *homogeneous* and *heterogeneous* computing architectures. Homogeneous computing, which uses one or more machines of the same type, has provided adequate performance for many applications in the past. Many of these applications had more than one type of embedded parallelism, such as single instruction, multiple data (SIMD) and multiple instruction, multiple data (MIMD). Most of the current parallel machines are suited only for homogeneous computing, however, numerous applications that have more than one type of embedded parallelism are now being considered for parallel implementations. As the amount of homogeneous parallelism in applications decreases, the distributed homogeneous system cannot offer the desired speedups. Therefore, a suit of heterogeneous architecture to exploit the heterogeneity in computations came to be a critical research issue [5]. Heterogeneous computing (HC) is the well-orchestrated and coordinated effective use of a suite of diverse high-performance machines, including parallel machines to provide super speed processing for computationally demanding tasks with diverse computing needs. Vision processing that has different computation requirements at each processing level can be an example of the effective use of heterogeneous computing [1]. HC systems include heterogeneous machines, high-speed networks, interfaces, operating systems, communication protocols, and programming environments, all combining to produce a positive impact on ease of use and performance. Heterogeneous computing should be distinguished from network computing or high-performance distributed computing, which have generally come to mean either clusters of workstations or ad hoc connectivity among computers using little more than opportunistic load-balancing. HC is a plausible, novel technique for solving computationally intensive problems that have several types of embedded parallelism. However, in order to best use heterogeneous computers to our advantage, a well constructed mechanism that orchestrates the heterogeneous computing resources effectively is essential, especially if the purpose of the system is to minimize total completion time (called "makespan"). Therefore, the problem of mapping and scheduling tasks to heterogeneous machines is important in order to maximize system throughput in an HC system [3][12][11].

The general problem of assigning independent tasks onto heterogeneous computers is known to be NP-complete. Thus, most of approaches so far have developed new heuristics, which have been specifically tailored to the problem [12][109][93][10][104][3]. Researchers have developed a variety of heuristic algorithms. However, often a heuristic, by its very nature, which works well in a specific environment does not work as well in another. This effect is amplified in a dynamic system where new machines may join or depart the system [3]. The performance of heuristic algorithms is influenced by dynamic system variables including system heterogeneity. Therefore, the choice of a mapping heuristic in a dynamic system environment remains a difficult problem.

In this research, rather than create another new heuristic, we develop a method of dynamically selecting a heuristic among a group of conventional heuristics. The conventional heuristics should be chosen in a way that is suited for the particular system. We propose a support vector scheduler (SVS) framework, which is based on support vector machine (SVM), that selects a heuristic from available heuristics to map independent tasks onto heterogeneous machines. In this dissertation, we use the finish time on the objective function; however, our research is easily extensible to other objective functions. We generalize the correlation between system variables and heuristics' performance through learning, resulting in a parameterized quadratic function, namely support vector (SV) model. The SVS is able to make a heuristic selection using the SV model. To the best of our knowledge, such an approach to heuristic selection for task mapping that accounts for dynamic system variables has not been studied before.

The remainder of the dissertation is structured as follows. Chapter 2 describes background. Chapter 3 discusses related work. In Chapter 4 we address PDC systems. Chapter 5 addresses heterogeneous computing. Chapter 6 introduces the SVM. Chapter 7 gives the descriptions of the design and mechanism of Particle Swarm Optimization & SVM^{pso}, which is implemented here. Chapter 8 presents the SVS framework. In Chapter 9 the simulation procedure is presented. Chapter 10 shows results with analysis. Chapter 11 has the concluding remarks.

Chapter 2

Background

In this chapter, we will introduce a variety of heuristic and machine learning algorithms to solve the task mapping problem.

2.1 Heuristic algorithm

First, heuristic algorithms can be categorized into batch mode and immediate mode algorithm. In immediate-mode, the task is mapped onto the machine immediately upon arrival, but in batch mode the task is not scheduled until a mapping event occurs.

2.1.1 Immediate mode mapping heuristics

Minimum completion time (MCT) heuristic is a variant of fast-greedy heuristic. It has been used as a benchmark for the immediate mode [3]. *MCT* assigns each task to the machine on which to complete the task the earliest. Braun et al. [20] compared 11 heuristic algorithms and found *MCT* to perform around the median of heuristics. *MCT* requires $O(m)$ time to find the machine that can finish a task earliest, where m is the number of machines. In minimum execution time (MET) heuristic, as a job arrives, each task is assigned to the machine that provides the least execution time for that task. Although *MET* heuristic is very simple with complexity $O(m)$, it may result in severe imbalance in load across the machines [20]. All of the computing nodes in the cluster are examined to determine the node that gives the best execution time for the job. As mentioned, *MET* may result in load imbalance at some point because it does not consider the ready time of each machine. With respect to this, switching algorithm (SA) alternates from *MET* to *MCT* at the sense of load imbalance. *SA* has the same complexity with *MCT* and its performance is close to *MCT*. K-percent best (KPB) heuristic implements the idea that too much selection pressure may lead to local optimum since it suppresses the diversity of search. The parameter K determines the selection pressure. Therefore, in *KPB*, a subset of machines, in which K is less than 100, is selected based on the earliest completion time. A task is assigned to the machine in the reduced set whose completion time is the least. That is, *KPB* looks forward to achieving the improvement in the long run as considering task heterogeneity instead of expecting the current marginal improvement promptly. In a similar way, feasible robust k-percent best (FRKPB) first finds the feasible set of machines for the newly arrived task. From this set, the *FRKPB* identifies the k-percent that has the smallest execution times for the task [30]. In opportunistic load balancing (OLB), which was known as a naive $O(n)$ algorithm, it simply places each job in order of arrival on the next available machine regardless of its completion time. The performance of *OLB* is worse than other aforementioned algorithms.

2.1.2 Batch mode heuristics

Whereas tasks are mapped onto machines immediately upon arrival in immediate mode, they are collected into a set that is examined for mapping at prescheduled times in batch mode. This enables mapping heuristics to make better decisions because the heuristics have the resource requirement information for a whole meta-task. When the task arrival rate is high, there is a sufficient number of tasks to keep the machines busy between the mapping events. Min-Min heuristic algorithm uses expected time to compute (ETC) matrix to make a decision for mapping a task to a suitable machine. The expected completion time is defined as:

$$C_{ij} = E_{ij} + R_j \tag{2.1}$$

The completion time of task i in machine j is calculated by adding the ready time of machine j to its expected completion time (2.1). Basically, a task is assigned to the machine that provides minimum completion time. When there is contention for the same machine to which two or more tasks are eligible, a task is assigned to the machine that should result in a change of the least amount of ready time, where ready time is the time when the machine becomes available after having executed the tasks previously assigned to it. In this algorithm, it is expected that a smaller makespan can be obtained if a larger number of tasks are assigned to the machine that, not only completes them earliest, but also executes them fastest. Max-Min heuristic is similar to *Min-Min* heuristic except that a task with maximum completion time is chosen among the candidate tasks whose completion time is minimum for all the machines. The *Max-Min* heuristic is likely to be better when there are more short tasks than long tasks since it can execute many short tasks concurrently along with the long task. The main idea of Sufferage heuristic is to assign a task to a machine that would suffer most if it is not assigned to the machine. The Sufferage algorithm also uses the same ETC matrix as it is used in *Min-Min* heuristic. An arbitrary task with ETC is selected and assigned to a corresponding machine. If the machine, however, is already assigned a task, an old task is replaced by a new one and returned to the task queue or it keeps its place by comparing between both sufferage values, which is the difference between the earliest completion time and the second earliest completion time.

2.2 GA(Genetic Algorithm)

Genetic Algorithms (GAs), devised by Holland [116], are adaptive heuristic search algorithms based on the evolutionary ideas of Darwin's principle of natural selection and genetics. In GA, a group of solutions evolve towards the good area in the search space. A population–(a group of individual solutions) evolves by surviving or dying through natural selection using various operators. In its' simple form, it recursively applies the concepts of *selection, crossover*, and *mutation* to a randomly generated population of promising solutions as described in Procedure 2. In [12] a GA was used to minimize the makespan, and the algorithm outperformed six other heuristic algorithms, about 10% against Early First algorithm.

Procedure 1 The Procedure of Genetic Algorithm

```
Initialize Population
Evaluate Population
while until stopping condition is met do
    Select parent
    Create offspring
    Evaluate offspring
    Select survivors
end while
```

2.3 Load sharing algorithm

In a heterogeneous computing environment with two processor classes, fast and slow, a job migration mechanism can be used. This mechanism distributes loads fairly over all processors from slow to fast using six scheduling strategies: probabilistic, probabilistic with migration of generic jobs, shortest queue (SQ), shortest queue with migration of generic jobs (SQM), least expected response time for generic jobs-maximum wait for dedicated jobs, least expected response time for generic jobs-maximum wait for dedicated jobs with migration[9]. In overall performance, SQ and SQM methods were better than all other methods.

2.4 Machine Learning

Scheduling plays an important role in production control in flexible manufacturing system (FMS), which involves several real-time decisions, such as part type and machine selection [13]. Consequently, a scheduled FMS is able to improve the machine utilization, enhance throughput, reduce the number of work-in-process, mean flow time, and the number of tardy parts. Assigning correct dispatching rules dynamically is critical for the scheduling problem. After receiving useful information from an FMS, a good scheduler should be able to make a right decision, i.e., output a right dispatching rule, for the next period to gain good performance. It needs as much expert knowledge stored in the scheduler as possible. Due to such reasons, Machine Learning, based on simulated sample data, has been used with good results.

Chapter 3

Related work

This chapter introduces conventional heuristics to solve the mapping problem and machine learning approaches to various scheduling problems as related to our work. Much work has been done in the study of the dynamic task mapping problem for the distributed computing systems [35][36][12][2][11][20][8]. A number of systems have been developed for managing the execution of tasks on a network of machines, scheduling tasks in order to evenly balance the load on the machines in the meta computer [32][33][34] [9][37][10]. The Switching Algorithm (SA) [3] utilizes the *MCT* and *MET* heuristics in an alternating manner, considering the load distribution across machines, which is determined by computing the load balance index based on machine ready times. However, the performance of *SA* is very close to *MCT*. The SmartNet scheduler [10] used a variety of scheduling algorithms to obtain near optimal schedules for different problems. However, the selection was done statically and was based upon whether the tasks had dependencies or not. It used *MinMin* and *MaxMin* heuristics when all of the tasks are computationally intensive and independent, and other heuristic algorithms when there are data dependencies among tasks. It does not select between *MinMin* and *MaxMin* heuristics. Evolutionary Techniques approach the task mapping problem by formulating the problem as an optimization problem in which an optimal or near-optimal solution is sought by exploring the solution space to find best available values of an objective function given a defined domain. Page & Naughton [12] used genetic algorithms (GAs) to dynamically schedule heterogeneous tasks on non-identical processors in a distributed system. However, the complexity of GA techniques is high and therefore they are difficult to use at run time.

Researchers have also attempted to apply machine learning technique to solve scheduling problems arising in a variety of domains. Liu, Huang, and Lin [13] used a multi-class SVM to schedule tasks in a flexible manufacturing system by dynamically applying a set of dispatching rules according to real-time system attributes. Their scheduler outperformed all static dispatching rules, with its mean throughput outperforming first-in-first-out scheduler by about 7%. Park, Casey, and Baskiyar [114] used a multi-class SVM to dispatch tasks dynamically onto heterogeneous machines by directly choosing appropriate machines to run tasks. The approach proved plausible, with performance very close to Early First [12] algorithm. However, it does not select heuristics, rather it directly maps tasks to machines –it can not be used to synthesize heuristics in the same way as the work in this research. It also functions in immediate mode whereas the research in this paper addresses tasks in batch mode.

Burke, Petrovic, and Qu [38] built a case-based reasoning (CBR) knowledge based system, to solve a timetabling problem, specifically a university course and exam timetabling system which has various constraints. Their system operates by memorizing heuristics that worked well in similar situations in a case base and retrieving the best heuristic for solving any new problem by searching the knowledge database. It provides better quality solutions, than a single heuristic by using different heuristics which were selected dynamically. In order to evaluate the quality of solutions, they used the following formula: $P = \sum_s W_s \times S_s$ where S_s is the number of situations in which violation of constraint s occurred and W_s is a weight that reflects the importance of constraint. We note that in the CBR System, a knowledge database is updated with new cases, whereas SVM does not require update until new training is needed. Arbelaez, Hamadi and Sebag [40] applied a combination of heuristics dynamically using SVM to find a solution to a constraint satisfaction problem (CSP). It outperformed a single heuristic solution in solving a collection of nurse scheduling problems by solving on average 50% more instances than a single heuristic. The CSP problem was solved by searching a tree using correctly chosen heuristics to find a solution. The selection of heuristic is based upon its execution time vis-a-vis a default heuristic. If worse, the default is used. The domain of this problem is different from the research presented in this paper. Also, the objective function in SVS is makespan and it schedules in batch mode rather than immediate mode, SVS also addresses issues of adaptability, task and machine heterogeneity and scalability.

Ravikumar and Vedi used a special class of neural networks, called Boltzmann Machines for solving the problem of statically mapping tasks onto processors in a reconfigurable environment [39]. The neural algorithm was much slower than heuristic algorithms, but provided superior solutions than the heuristic algorithms. Their neural networks rely on the principle of empirical risk minimization, whereas our approach develops a meta-scheduler based on the principle of structural risk minimization making the latter adaptive. Beck and Siewiorek represented a task allocation problem for bus-based multicomputers as a vector packing problem. The goal was to obtain an assignment of tasks to nodes that is feasible in respect to the constraints that minimize the number of processing nodes and the utilization level of the broadcast bus. It was also further formulated as a multi-dimensional problem [91]. The task allocation problem was formulated as the minimization of a quadratic pseudo-boolean function with linear constraints [92]. Hong and Prasanna scheduled a large set of equal-sized independent tasks on heterogeneous computing systems to maximize the throughput of the system by using an extended network flow representation [93]. Khalifa et al. created utl*MinMin* heuristics, which revised an original *MinMin* heuristics, that employs preemptive mapping approach that allows tasks in execution to be interrupted, moved to another processor, and resume execution in a different environment [94]. This algorithm was motivated by the fact that the original *MinMin* algorithm can cause some tasks to be starved of machines due to the expected heterogeneous nature of the tasks. That is, some tasks may be remapped at each successive mapping event without actually beginning execution. Meanwhile, other machines may remain idle during the whole mapping session. Braun et al. discussed the factors that makes it difficult to select the best heuristic in a given scenario in a heterogeneous computing (HC) environment [11]. In [96], a heuristic algorithm based on genetic algorithm for the task mapping problem was used in the context of local-memory multiprocessors with a hypercube interconnection topology. Chen et al. solved a task scheduling problem in grid environment as a task-resource assignment graph and thus mapped the task scheduling problem into a graph optimal selection problem. They solved the optimization problem using Particle Swarm Optimization [97]. Huang et al. solved a task mapping problem as Master-Slave model in grid environment, where the master node delivers a set of equal-sized and independent tasks to available slave nodes in a single-port pattern which means tasks are transmitted sequentially. Each slave begins processing tasks after receiving them, and then returns computation results upon completion[98]. In [95], the mapping problem was solved based on randomized mapping whose key idea is to randomly choose a valid mapping event and evaluate its performance. By repeating this process a large number of times and picking the best solution that has been found over all iterations, it is possible to achieve a good approximation to the global optimum. The intuition behind this is that any algorithm that does not consider all possible solutions with a non-zero probability might get stuck in a local optimum. With the randomized approach any possible solution was considered and chosen with a small, but non-zero probability.

Chapter 4
Parallel and Distributed Computing (PDC)

Although the essence of internal design of computer and the way information flows within a computer have not changed since von Neumann computer [60], noticeable advancements in computing technology have been made by the changes in implementation from vacuum tubes to integrated circuits, and further the availability of fast, reliable, and cheap electronic components, thus bringing revolutionary changes in computing paradigm. These resolute developments in computing technology enabled the solution of a wide range of computationally intensive problems requiring great computational speed such as numerical modeling and simulation of scientific and engineering problems [58][59]. However, whatever the computational speed of current processors, there are still many problems not possible to process by single processor. For example, the SETI@home project [56] is the first attempt to use large-scale distributed computing with over 3 million users to perform a sensitive search for radio signals from extraterrestrial civilizations. Radio telescope signals consist primarily of noise (from celestial sources and the receiver's electronics) and man-made signals such as TV stations, radar, and satellites. Modern radio SETI projects analyze the data digitally. More computing power enables searches to cover greater frequency ranges with more sensitivity. Radio SETI, therefore, has an insatiable appetite for computing power.

It is evident that the performance of single processor computer is limited, because processor has its clock speed constrained by the limitation of device technology such as CPU power and dissipation problem resulting in thermal hotspot by non-uniform temperature distribution across processor chip [21]. Therefore, the evolutionary transition from sequential to parallel and distributed computing (PDC) is seen as inevitable flow in solving computing-power demanding problems to overcome the limitation from the use of a single processor.

Furthermore, PDC system is recognized as an important means for the solution of many grand challenge problems [61] that refer to really difficult tasks that stretch the limits of cognitive ability in science and engineering. These include climate modeling, human genome mapping, semiconductor and superconductor modeling, pollution dispersion, and pharmaceutical design. It is common that these applications require computing power greater than that is obtainable with many conventional computers. These applications are common to exceed the computing power which is obtainable with many traditional computers. Although this technology has risen to prominence during last decades, many relevant issues still remain unresolved. The field is in a state of rapid flux where various applications are being made on several subsequent issues. In this chapter we will review relevant subjects that are essential to make it useful to use PDC systems.

4.1 Von Neumann computer

The overwhelming majority of today's computers follow architectures and modes of operation, the same basic design principles formulated in the late 1940s by John von Neumann and coworkers [62]. The central idea of von Neumann computer is that it fetches an instruction and its operands from a memory unit, and saves in memory the results from the execution of the instruction in the processing unit. The von Neumann architecture is depicted in the Fig. 4.1. In such a computer, the instructions are executed sequentially, one operation at a time. One problem of conventional von Neumann machines is *CPU to memory* bottleneck, which mainly results from the disparity between high CPU speeds and much lower memory speeds. Over time, efforts to improve the early designs techniques were introduced, such as *cache memories* and *pipelining* [64] [65]. Although these efforts concentrated on the improvement of single processor mechanism, the development of parallel and distributed computing is slowly leading to the end of sequential processing depending on the performance of single processor.

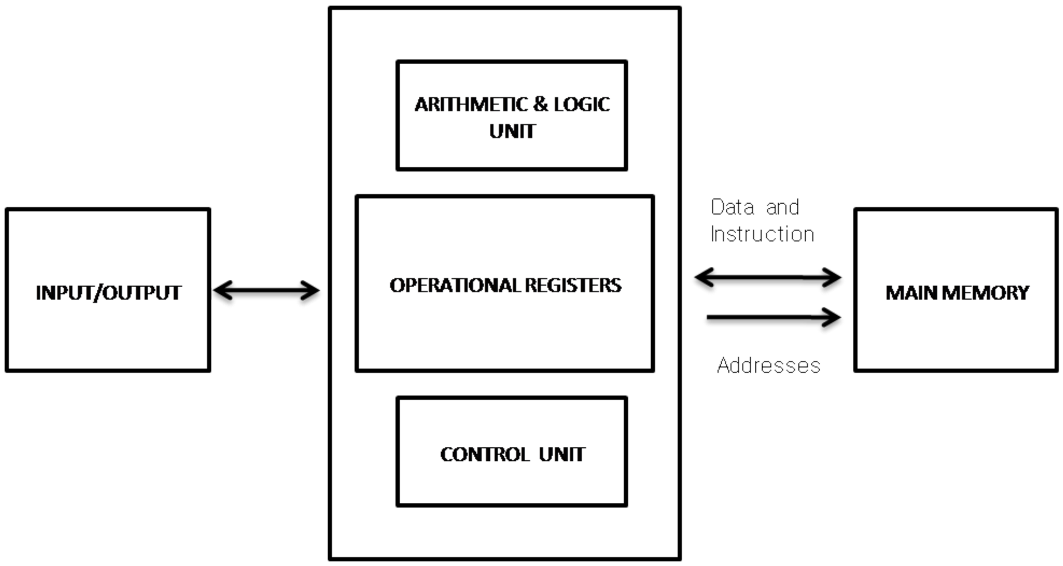

Figure 4.1: The von Neumann computer

4.2 Parallel Processing Paradigms

Computers operate simply by executing instructions on data. A stream of instructions informs the computer of what to do at each step. Flynn [44] created a classification for the architecture of a computer on the basis of how the machine relates its instructions to the data being processed. The multiplicity of instruction streams and data streams in the system produced following four classifications:

- single instruction stream, single data stream (SISD)
- single instruction stream, multiple data stream (SIMD)
- multiple instruction stream, single data stream (MISD)
- multiple instruction stream, multiple data stream (MIMD)

All von Neumann machines belong to the SISD class. A SIMD machine consists of N processors, N memories, an interconnection network, and a control unit. All the same processing elements are supervised by the same control unit, and the processors operate on different data sets from distinct data streams. For example, for a SIMD computer with

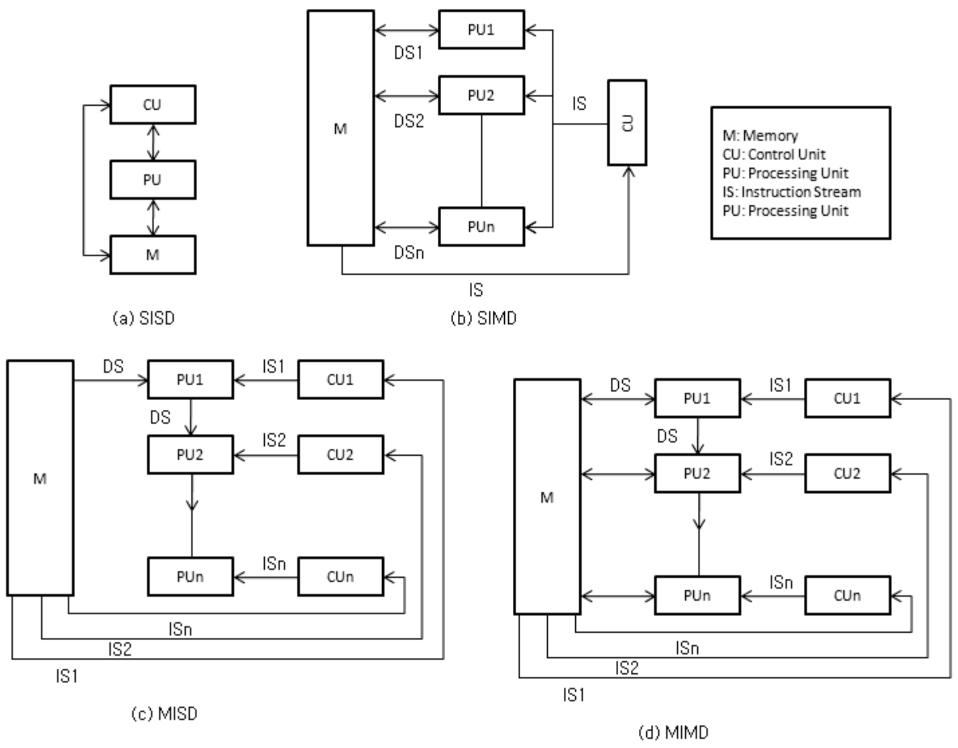

Figure 4.2: Flynn's taxonomy

N processors, each processor will be executing the same instruction at the same time, but each on its own data. SIMD machines can be classified into two categories: *shared-memory* (SM) and *local memory* (LM) [66]. High performance on SIMD machines requires rewriting of conventional algorithms to manipulate many data simultaneously by sending instructions to all processors. SIMD machine can solve efficiently the problems that require parallel manipulations of large data sets, for example, algorithms based on vector and matrix operations [78] [79], and image processing algorithms where operations are performed on a large number of image pixels [80] [81] [1]. Furthermore, many large-scale SIMD parallel machines have been implemented including:

- Illiac IV: Illiac IV SIMD machine was implemented with 64 processing elements (PEs) with 64-bit words, an N-bit vector is used as the mask. Each vector bit represents the state of one PE. If the bit is a 1, the corresponding PE will be active; otherwise, the

PE will be inactive. While this scheme permits enabling and disabling of arbitrary PE sets, it is prohibitively expensive for large N. It was used to run large-scale computations needed for climate simulations, signal processing, seismic research, and physics calculations.

- *Massively Parallel Processor* (MPP): MPP, which incorporates 16,384 bit-serial PEs, was designed for high-speed satellite imagery processing [82] , and was delivered to NASA in 1982.

- The Connection machine CM-2: A fully configured CM-2 machine comprises 65,536 PEs that are divided into four groups. Each group of 16,384 PEs is controlled by a *sequencer*, and the four sequencers are connected up to four front end computers via a 4 x 4 *Nexus* switch.

- The GF11 parallel computer: The GF11 SIMD machine, designed at the IBM T.J. Watson Research Center, contains 566 very powerful processing elements.

- The MasPar MP-1 and MP-2: The machine built by MasPar Computer Corporation contains up to 16,384 processing elements. The two machine types have the same basic architecture and differ mostly in the processing element complexity; While the PEs in the MP-1 are 4-bit wide, those of the MP-2 are 32-bit CPUs.

In a MISD computer, each of the processors has its own control unit and shares a common memory unit where data reside. Therefore, parallelism is realized by enabling each processor to perform different operations on the same datum at the same time. Systolic arrays are known to belong to this class of architectures [67] [68]. In MIMD machines there are N processors, N streams of instructions, and N streams of data. The processors of MIMD machines are of the same type used in a SISD computer; that is each processor has its own control unit in addition to its local memory and arithmetic and logic unit. A MIMD computer is considered tightly coupled or loosely coupled according to the degree of interactions among processors. As an addition to Flynn's taxonomy, another class known as single program, multiple data (SPMD) can be included within the MIMD classification to describe the cases where many programs that have the same process type are executed on different data sets, synchronously or asynchronously, e.g., a single source program is written and each processor will execute its personal copy of this program [69].

4.3 Organization of PDC systems

PDC systems can be organized in two folds: General purpose architecture and Special-purpose architecture. Theoretically, PDC systems can be constructed with a general purpose by building parallel computers for a wide variety of applications based on the MIMD model. In practice, it is reasonable to assemble several processors in a configuration specifically designed for the problem under consideration. Fig. 4.3 provides an overall organization of parallel computers. In SIMD machines, the parallel operations are synchronized at the machine language level, and scheduling and allocation need to be done by the programmer. In MIMD machines, the processes that run in parallel need to be synchronized whenever they communicate with each other. In general, paradigms for building parallel computers can be divided into the following categories:

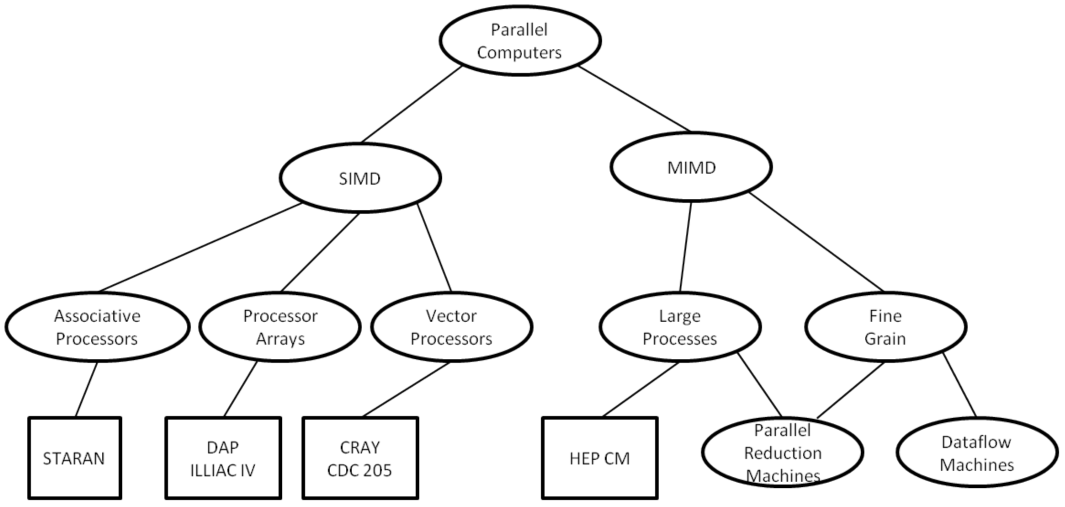

Figure 4.3: The organization of parallel computers

- *Pipelining* : This is a classical way to exploit parallelism and concurrency in computer architectures. *Pipelining* was stimulated by the need for faster and more cost-effective systems. It refers to a segmentation of a computational process. Ramamoorthy et al. [70] defined this technique as organizing repeated sequential processes in an overlapped mode. In a pipelining architecture, computational processes are decomposed into several sub-processes, and processed in an overlapped mode by dedicated autonomous modules. As an illustration, consider the process of executing an instruction in computers. Processing instructions involve several repetitive processes: Fetching, Decoding, and Execution. Thus, each process can be pipelined by a dedicated unit like an industrial assemble line.

- *Systolic Array processors*: The word *systolic* has been borrowed from the medical field;-just as the heart pumps blood, information is pumped through a systolic array in various directions, and at regular intervals. These are systems that consists of a number of processing elements with nearest-neighbor connection that operate in parallel under the direction of a single control unit. Array processors can be viewed as a subclass of SIMD computers in that each of the processing elements performs the same operation at the same time, on different data elements. Fig. 4.4 shows a systolic array used to multiply 4 x 4 matrices, A and B. The elements of A enter from the left and the elements of B enter from the top. The final product terms of C will be held in the processors as shown. Each processor, $P_{i,j}$, repeatedly performs the same algorithm on different data elements.

- *Vector processors*: A vector processor is a processor that can operate on an entire vector in one instruction. The operands to the instructions are complete vectors instead of one element. Therefore, vector architectures exhibit SIMD behavior by having vector operations that are applied to all elements in a vector. Most vector machines have

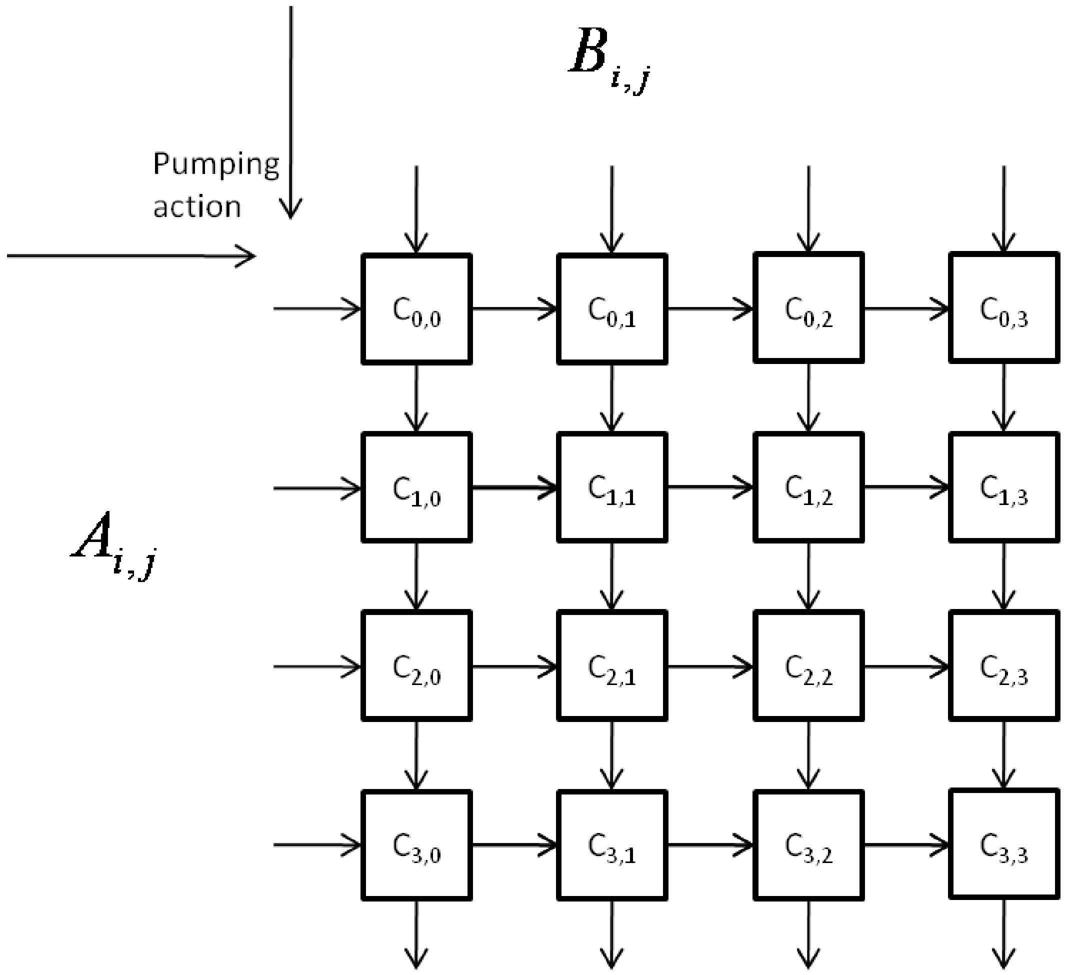

Figure 4.4: Matrix multiplication using a systolic array

a pipelined structure and specially designed register sets (vector-register architecture) that can hold vectors.

- *Multiprocessors*: These are small scale MIMD machines. A single processor model is extended to have multiple processors connected to multiple memory modules, such that each processor can access any memory module in a so-called *shared memory* configuration, as shown in Fig. 4.5. This architecture, which provides processors with *uniform memory access* (UMA) structure, is attractive because of the convenience of sharing data. However, it is especially difficult to implement the hardware to achieve fast access to all the shared memory by all the processors. Hence, most large, practical shared model systems have some form of hierarchical or distributed memory structure, such that processors can access physically nearby memory locations much faster than more distant memory locations, thus resulting in *non-uniform memory access* (NUMA) structure. The NUMA model is described in Fig. 4.6. *Message-passing multiprocessors*, which is an alternative form of multiprocessor system, is created by connecting complete computers through an interconnection network, as shown in Fig. 4.7. In this architecture, a processor can only access a location in its own memory, thus the interconnection network is provided for processors to send messages to other processors.

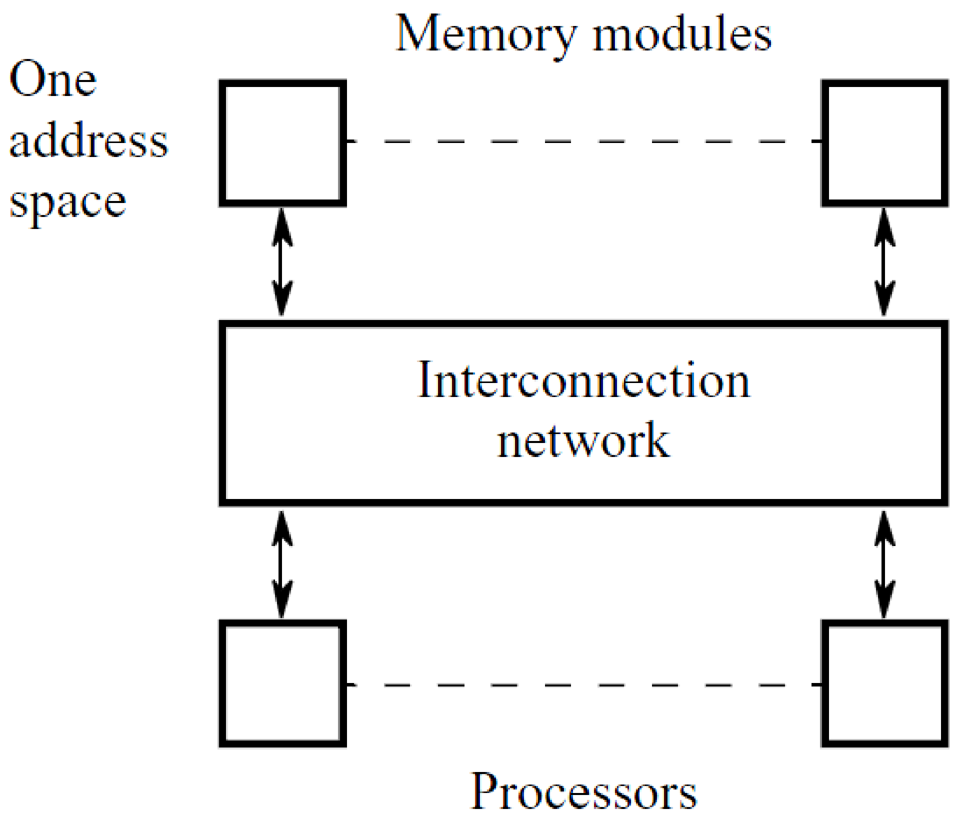

Figure 4.5: Traditional shared memory multiprocessor model

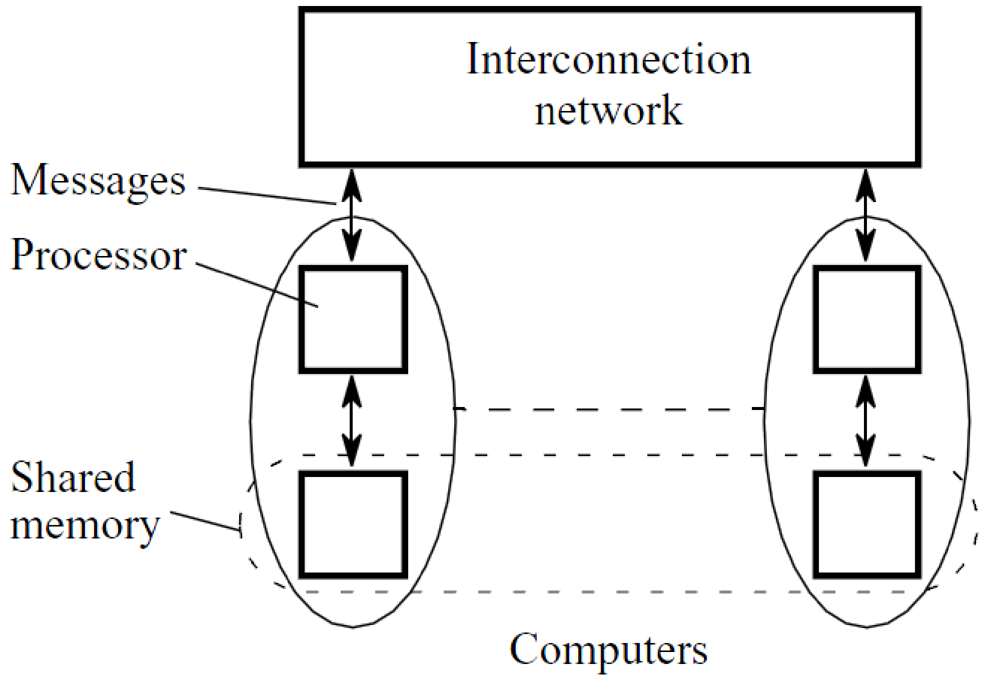

Figure 4.6: Shared memory multiprocessor implementation

- *Dataflow machines & Reduction machines*: Dataflow machines directly contrast with the traditional von Neumann architecture in that instructions are executed in out of order without both control flow and program counter. Normally instructions are sequentially organized without recoding dependency information between them into binaries. It is because binaries compiled for dataflow machine contain this dependency information that enables out-of-order execution. Therefore, in dataflow machines scheduling is based on the availability of data. Reduction machines are different from dataflow machines in that scheduling is based on the need. Therefore, reduction machines are known as *demand-driven* execution model, while dataflow machines are *data-driven* execution model [71].

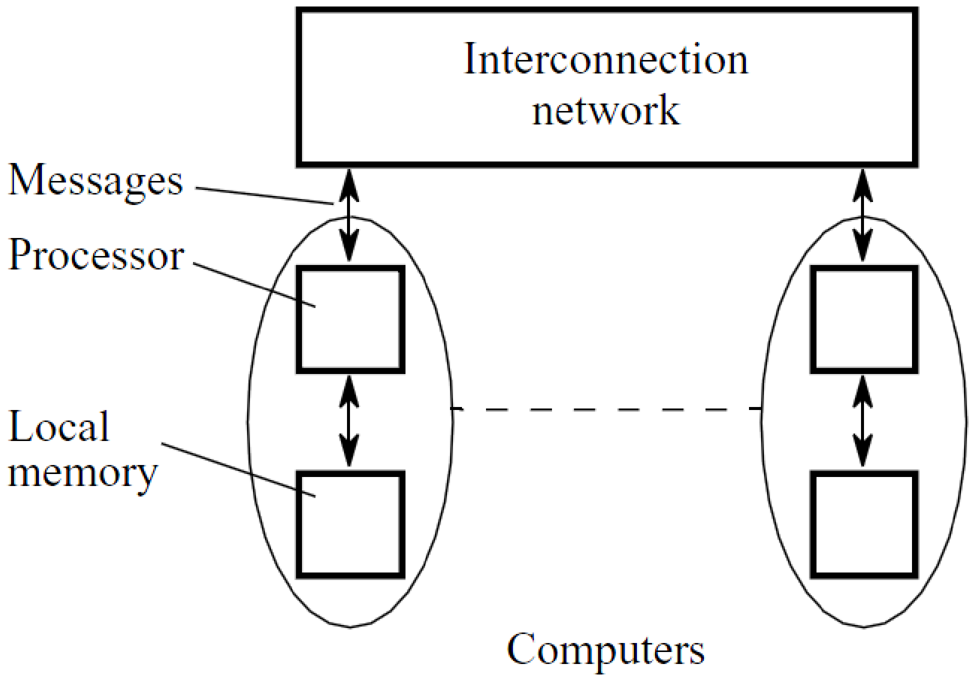

Figure 4.7: Message-passing multiprocessor model(multicomputer)

- *Neural networks*: Neural computers learn or model the behaviors of complex systems through the use of a large number of processors and interconnections.

- *Clustering*: This is a paradigm that enables multiple computers to cooperate simultaneously to solve a computationally intensive problem. The rapid increase in microprocessor performance and network bandwidth has made clustering a practical, cost effective computing solution which is readily available to the masses. At the hardware level, a cluster is simply a collection of independent systems, typically workstations, connected via a commodity network. Cluster computers communicate through software methods such as message passing and distributed shared memory. Mega problems, such as *grand challenge* applications, that require all the computational capabilities of any available system, including memory and CPU, historically have been used as the principal justification for adoption of massively parallel processing (MPP) on such *capability demand* problems. On the other hand, clusters are considered the most cost-effective solution for *capacity demand* problems requiring substantial, but far from ultimate, performance and making moderate demands on memory.

PDC systems are created by connecting multiple computers, using various forms of local area networks (LANs) and wide area networks (WANs) to provide high performance that approaches supercomputer levels [72]. The type and topology of inter connection network, therefore, is an important design issue in parallel processor and multicomputer systems. Many network topologies, such as bus architectures, ring networks, crossbars, meshes, shuffle exchanges network, and hypercube ensembles, are described in the literature [73]. Choosing an appropriate parallelizing algorithms that can make excellent use of a certain network topology is essential for achieving high performance, as well as the availability of more efficient and reliable networks. Important factors that characterize a parallel algorithm in relation to the host parallel architecture are the number of processors, capabilities of these processors, and so forth. In addition to a parallel algorithm, a parallel architecture is characterized by the control mode in which computers within the architecture operate. In most computers, the control mode is command driven, which means that different events are driven by the sequence of instructions. Other computers employ a data-driven approach (e.g., data flow machines). In this case, the control-flow mechanism is triggered by the availability of data. Another mode of control is demand driven, whereby computations take place only if their results are requested by other events. Alternate modes of control are based on combinations of these approaches.

4.4 PDC Systems performance

With respect to system performance, PDC systems have several issues to be solved in relation to parallelization that do not arise in sequential systems. One of these issues is *task allocation*, which is the breakdown of the total workload into smaller tasks assigned to different processors, and the proper sequencing of the tasks when some of them are interdependent and cannot be executed simultaneously. A PDC system can achieve the highest level of performance when it sustains high per-processor utilization through the process of proper scheduling or load balancing. The scheduling problem belongs to an NP-complete problem [74] [75] [76] [77].

4.5 Partitioning and Scheduling

Partitioning deals with how to detect embedded parallelism in the program. For this, the program is first analyzed to determine the *ideal parallelism* revealed by the control and data dependences. Two operations which are not related by control or data dependences can potentially be executed in parallel. This kind of parallelism is usually referred to as *fine grain parallelism* because the unit of parallelism is normally an operation or a single statement. Several operations or statements may be combined into a larger grain to reduce communication overhead. The problem is to find the best grain size that maximizes parallelism while reducing overhead. In the context of parallel and distributed computing, partitioning must be followed by scheduling in which the concurrent parts of a parallel program are arranged in time and space so that the overall execution time of the program is minimized. The problem of scheduling program tasks onto multiprocessor systems is known to be NP-complete in general. The intractability of the scheduling problem has led to a large number of heuristics, each of which may work under different circumstances. Partitioning and scheduling can be conducted *implicitly* or *explicitly*. In the implicit approach, compilers are required to analyze the application to explore embedded parallelism and perform partitioning, for example, when the underlying computing environment is entirely concealed from the programmer. On the other hand, in the explicit approach, the programmer is responsible for identifying parallelism within the application. In this case, existing languages are extended, or entirely new languages are required to express parallelism directly.

4.5.1 Dynamic scheduling

Dynamic scheduling is any runtime technique for placing tasks onto processors and deciding when is best to execute them so that parallel programs finish in the earliest possible time. The most elementary approach to dynamic scheduling attempts to balance processor load across m processors using very local information. In its simplest form, $m + 1$ processors are used: one processor runs a scheduler that dispatches tasks on a first-in-first-out (FIFO) basis to all other m processors as shown in Fig. 4.8. Each of the m processors maintains a private list of waiting tasks called the *waiting queue*. This FIFO list holds all tasks assigned to the processor. As soon as one task is finished, the processor takes the next waiting task from its queue and processes it to completion. As tasks are processed, they may require the services of other tasks. Thus, requests for new tasks are made by the m processors as they do their work. These requests are placed on the *scheduler queue*, maintained by the scheduler processor. The scheduler processor also dispatches tasks in first-come-first-severed order. However, deciding which waiting queue to select is made by various heuristics. The simplest heuristic attempts to balance the load on all processors.

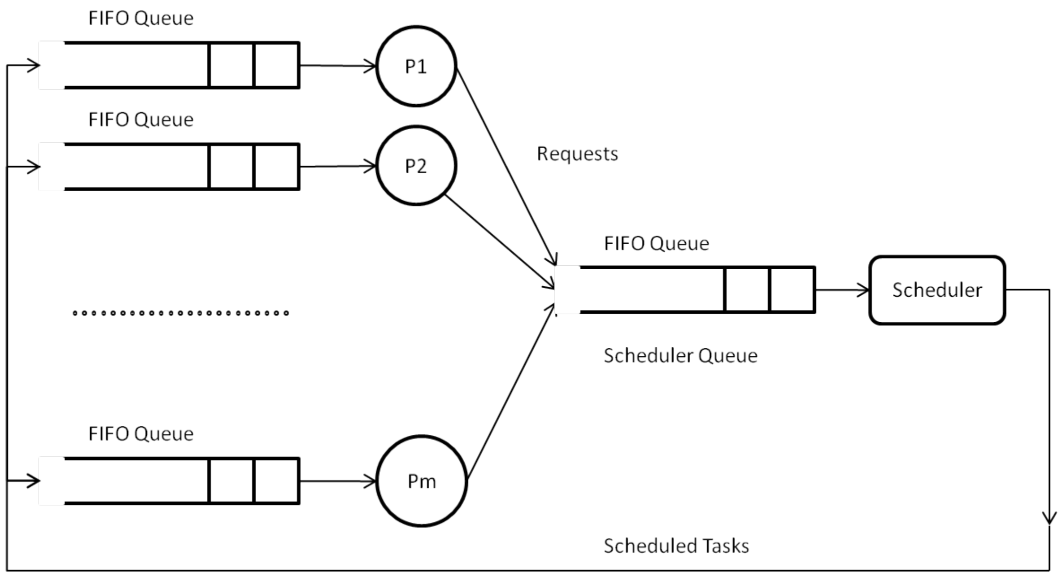

Figure 4.8: Dynamic scheduling

4.5.2 Static scheduling

Static scheduling of nondeterministic programs requires the use of a different program model that distinguishes between conditional branching and precedence relations among parallel program tasks. The branch graph represents the control dependences and the different execution instances of the program, while the precedence graph represents the data dependence among tasks. An execution instance of a program can be defined as the set of tasks that are selected for execution at one time for some input. For scheduling, all execution instances within a program are first defined by the branch graph. Precedence graphs are used to construct task graphs based on execution instances from the branch graph, each of which consists of the nodes given in the corresponding execution instance and the precedence relations among them. Each task graph also shows the amount of computation needed at each node as well as the size of the data messages passed among the nodes. Each task graph is scheduled independently using one of the techniques used in scheduling branch-free task graphs. Given a task graph and a target machine description, a schedule is created in the form of Gantt chart [100]. Finally, a number of Gantt charts are combined into a unified schedule. The schedule is given in the form of processor allocation and execution order of the tasks allocated to the same processor.

4.5.3 Task Allocation

In a distributed computing system made up of several processors, the interacting tasks constituting a distinguished program must be assigned to the processors so as to make use of the system efficiently. To improve the performance of a distributed system, it is necessary that task allocation be performed with minimal inter-processor communication between tasks and the execution cost. Thus, the purpose of task allocation technique is to find some task assignment in which the total cost due to inter-processor communication and task execution is minimized.

Chapter 5
Heterogeneous Computing

Homogeneous computing usually uses one mode of parallelism in a given machine (like SIMD, MIMD, or vector processing) and thus cannot adequately meet the requirements of applications that require more than one single type of machine. Therefore, homogeneous systems consisting of a single type of machine architecture might suffer from inherent limitations. For example, vector machines employ interleaved memory with a pipelined arithmetic logic unit, leading to performance in high million floating-point operations per second (Mflops). However, if the data distribution of an application and the resulting computations cannot exploit these features, the performance degrades severely.

In general, it is known that a single machine architecture is not able to satisfy all the computational requirements of various subtasks in certain applications in a feasible amount of time[24]. Thus, the limitation of homogeneous computing from the use of single machine architecture can be overcome by constructing a heterogeneous computing environment. A *heterogeneous computing* (HC) system provides a variety of architectural capabilities, orchestrated to perform an application whose subtasks have diverse execution requirements. Fig. 5.1 shows an example of a heterogeneous computing environment. HC systems can be constructed with two different modes. A *mixed-mode* HC system is a single parallel processing machine that is capable of operating in either the synchronous SIMD or asynchronous MIMD mode of parallelism, and can dynamically switch between modes at instruction-level granularity. In a *mixed-mode* the HC system, a single machine operates with multiple-mode, dynamically switching between different modes at instruction-level granularity with generally negligible overhead [84]. In a A *mixed-machine* mode, HC system consists of a heterogeneous suite of independent machines of different types interconnected by a high-speed network [85].

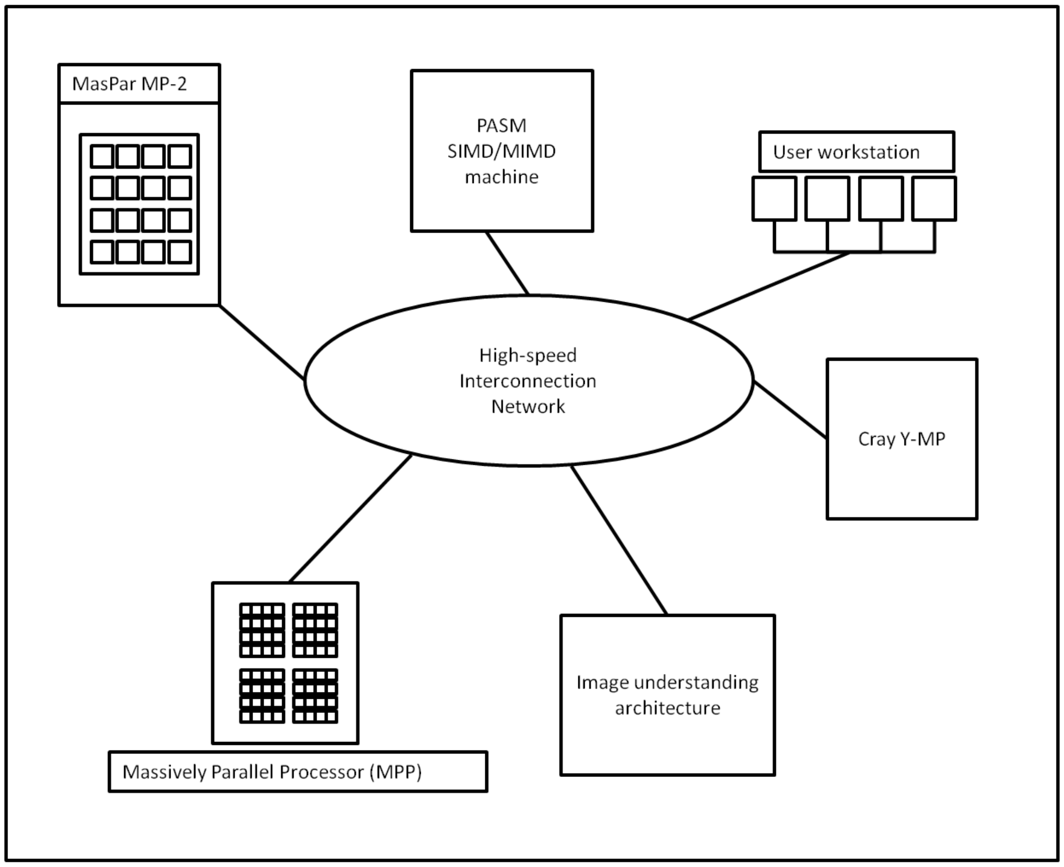
MasPar MP-2
PASM SIMD/MIMD machine
User workstation
High-speed Interconnection Network
Cray Y-MP
Image understanding architecture
Massively Parallel Processor (MPP)

Figure 5.1: An example heterogeneous computing environment

To fully exploit HC systems, a task must be decomposed into subtasks, which may have different machine architectural requirements. Fig. 5.1 helps support the notion that the system performance in a system running a hypothetical application which contains diverse computational types is distinctly different according to the system architecture. Executing the whole program on a vector super computer only gives twice the performance achieved by a baseline serial machine due to the speedup of the vector portion of the program, but slight improvement in the non-vector portion. However, the use of five different machines, each matched with the computational requirements of the subtasks for which it is used, can execute 20 times as fast as the baseline serial machine.

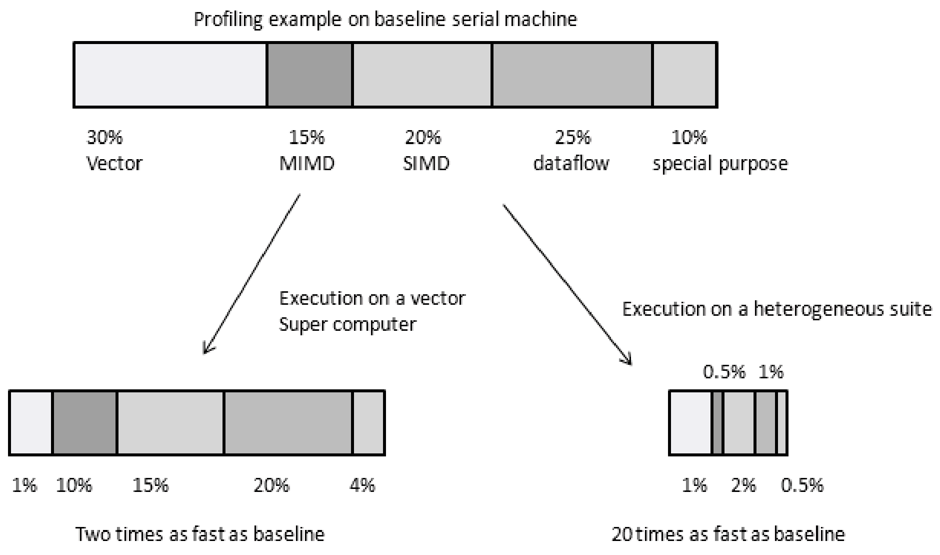
Profiling example on baseline serial machine
30% Vector
15% MIMD
20% SIMD
25% dataflow
10% special purpose
Execution on a vector Super computer
Execution on a heterogeneous suite
1%
10%
15%
20%
4%
Two times as fast as baseline
0.5% 1%
1% 2% 0.5%
20 times as fast as baseline

Figure 5.2: A hypothetical example of the advantage of using heterogeneous computing

5.1 Heterogeneous Systems

MIMD and SIMD machine are the most common heterogeneous systems [86].

5.1.1 Heterogeneous problem model

To exploit the HC system, application tasks should be decomposed into subtasks. Fig. 5.3 has described the decomposition of the task into subtasks, code segments, and code blocks. The parallel task T is divided into subtasks t_i, $1 \leq i \leq N$. Each subtask t_i is further divided into code segments t_{ij}, $1 \leq j \leq S$, which can be executed concurrently. Each code segment within a subtask can belong to a different type of parallelism (i.e., SIMD, MIMD,vector, and so forth) and should be mapped onto a machine with a matching type of parallelism. Each code segment may further be decomposed into several concurrent code blocks with the same type of parallelism. These code blocks t_{ijk}, $1 \leq k \leq B$, are suited for parallel execution on machines having the same type of parallelism.

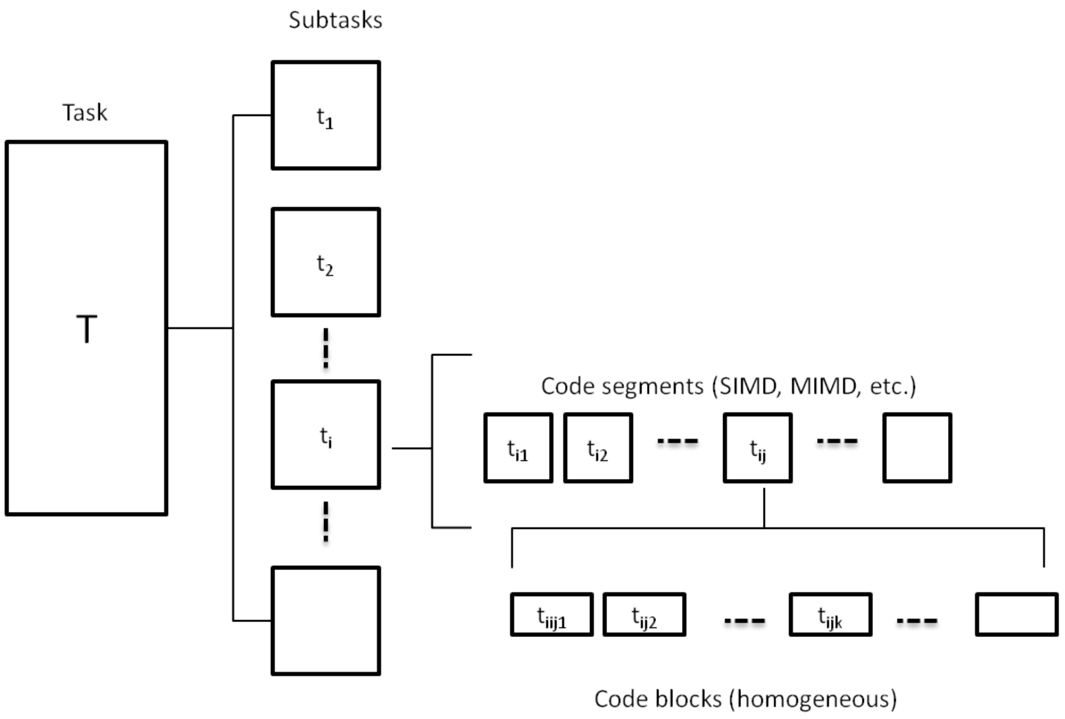

Figure 5.3: Heterogeneous application model

Table 5.1: Trade-offs between SIMD and MIMD

Operation	SIMD	MIMD
Synchronization overhead	No	Yes
Data parallelization	No	Yes
Control parallelism	No	Yes
Memory cost	Low	High

5.1.2 SIMD VS MIMD mode

A comparison of SIMD and MIMD machines is shown in Table 5.1. In SIMD mode, processing elements are implicitly synchronized at the instruction level. While implicit synchronization scheme is required in MIMD mode, it generally incurs overhead. MIMD allows different operations to be performed on different processing elements simultaneously through multiple threads of control, thus enabling independent *functional parallelism*, while the SIMD machine might result in time-delay for a sequence of data-dependent instructions due to the lack of concurrent operations.

5.2 Task Profiling and Analytical Benchmarking

To execute a task on a mixed-machine HC system, the task must be decomposed into a collection of subtasks, where each subtask is a homogeneous code *block*, such that the computations within a given code block has similar processing requirements (see Fig. 5.3). These homogeneous code blocks are then assigned to different types of machines to minimize the overall execution time. *Task profiling* is a method used to estimate the code's execution time on a particular machine and also to identify the code's type [87]. Code types that can be identified include vectorizable, SIMD/MIMD parallel, scalar, and special purpose (such as fast Fourier transform). Traditional program profiling involves testing a program assumed to consist of several modules by executing it on suitable test data. The profiler monitors the execution of the program and gathers statistics, including the execution time of each program module. This information is then used to modify the modules to improve the overall execution time.

Analytical benchmarking is a test procedure that measures how well the available machines in the heterogeneous suite performs on a given code-type [87]. While code-type profiling identifies the type of code, analytical benchmarking ranks the available machines in terms of their efficiency in executing a given code type. Experimental results obtained by analytical benchmarking show that SIMD machines are well suited for operations such as matrix computations and low-level image processing, while MIMD machines are most efficient when an application can be partitioned into a number of tasks that have limited intercommunication [5]. It can be noted that analytical benchmark results are used in partitioning and mapping.

5.3 Matching and Scheduling for HC systems

For HC systems, *matching* involves deciding on which machines each code block should be executed, and scheduling involves deciding when to execute a code block on the machines in which it was mapped. Mapping and scheduling for general distributed computing systems has focused on how to effectively execute multiple subtasks across a network of sequential processors. In such an environment, a load balancing scheme can be an effective way to improve system throughput. However, there is a fundamental distinction between mapping and scheduling for distributed systems consisted of sequential processors and mapping and scheduling for an HC systems consisting of various types of heterogeneous computers. In the latter case, a subtask may execute most effectively on a particular type of parallel architecture and matching subtasks to machines of the appropriate type is a more important factor than merely balancing the load across systems.

Matching and scheduling also can be viewed as a mechanism which is used to efficiently and effectively manage the access and use of a resource by its various consumers under specific policies [88]. In the context of HC systems, the *consumers* are represented by the code blocks. The *resources* include the suite of computers, the network that interconnects these computers, and the I/O devices. The *policy* is the set of rules used by the scheduler to determine how to allocate resources to consumers based on knowledge of the availability of the resources and the suitability of the available resources for each computer.

Chapter 6
Support vector machine

In this chapter, we introduce the central ideas of Support Vector Machine (SVM) and describe the support vector machine-particle swarm optimization (SVM^{pso}) we implemented using the Java programming language.

6.1 Overview of Support Vector Machine

In this section, we present an overview of SVM, as related to this research, for solving a binary classification problem. The SVM was originally proposed as a statistical learning theory based on the principle of structural risk minimization, and has recently gained wide acceptance especially in the area of pattern recognition [14], [19]. An example application is recognizing a pattern in text documents, e.g., classifying text documents automatically into pre-defined categories on the basis of previously seen patterns [102]. SVM is basically a supervised learning method in which samples are required to learn a specific pattern. A sample x_i and its label y_i are usually represented as a tuple (x_i, y_i), where $x_i \in \mathbb{R}^N$ has N-dimensional attributes $y_i \in \{-1, 1\}$, $i = 1, \cdots, l$, and l is the number of samples. (A list of relevant symbols has been presented in Table 6.1 for quick reference.) Learning from these samples is to find a general function, which best estimates the pattern of the samples, $f : \mathbb{R}^N \to \{-1, 1\}$ which can classify unknown samples (x, y) into +1 or -1, which are generated from the same underlying probability distribution $P(x, y)$. Such general functions can be expressed in the hyperplane form as:

$$(w \cdot x) + b = 0, \quad w \in \mathbb{R}^N, b \in R \tag{6.1}$$

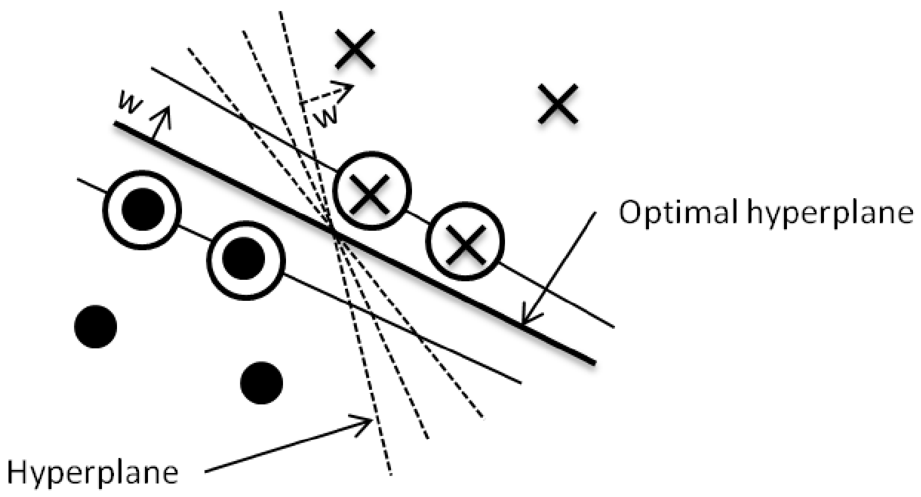

Figure 6.1: Example of large margin hyperplane with support vectors circled

where the vector w is orthogonal to the hyperplane, and varying b moves the hyperplane along w. To classify an unknown input vector x, in a binary fashion, we define function f, as simply yielding the sign (+ or -) of the function defined above:

$$f(x) = sign((w \cdot x) + b) \tag{6.2}$$

Fig. 6.1 shows the hyperplanes separating two distinct classes of samples represented by dots($\odot$) and crosses ($\otimes$). The support vectors are circled. There may exist infinite number of hyperplanes separating the samples. The training of SVM is to find the optimal hyperplane among these hyperplanes. The optimal hyperplane yields the maximal margin between two distinct classes of samples, and is parallel to the two hyperplanes consisting of support vectors as shown in Fig. 6.1. The optimal hyperplane can be directly found within the input space if the samples are separable, otherwise in higher dimensional feature space where they may be easily separable. Whether or not samples are separable is determined by the distribution of samples. If the non linear samples are inseparable in the input space, we first transform the coordinates via a mapping function ϕ to higher dimensional feature space as shown in Fig. 6.2, and then find the optimal hyperplane in the feature space. In the feature space, the dots and crosses represent the '+' or '-' outcomes of the evaluation of f in equation (6.2). In

Table 6.1: List of relevant symbols

ϕ	Function mapping from input to feature space
α	Lagrange multiplier
K	Radial Basis Function Kernel
x	Input vector
y	Label for input vector
n	Number of tasks
X	Support vectors
s	Number of support vectors
l	Number of samples
m	Number of machines
p	Number of tasks within a batch
q	Number of batches
β	Heterogeneity of negative samples
γ	Heterogeneity of positive samples

order to find the optimal hyperplane we have to solve the constrained optimization problem [19] which maximizes $||w||$ subject to:

$$\forall i \;\; y_i(x_i \cdot w + b) \geq 1 \tag{6.3}$$

Using Lagrangian, one can reformulate this optimization problem as a convex optimization problem, which consists of minimizing L_D with respect to α:

$$L_D = \frac{1}{2}\sum_{i,j=1}^{l} \alpha_i \alpha_j y_i y_j x_i \cdot x_j - \sum_{i=1}^{l} \alpha_i \tag{6.4}$$

subject to the constraint:

$$\sum_{i=1}^{l} \alpha_i y_i = 0 \tag{6.5}$$

where x_i represents the sample vector, y_i its class, α_i the Lagrange multiplier of x_i, and C $(0 \leq \alpha_i \leq C)$ is a parameter that determines the upper boundary for training errors (l is the number of samples, as stated earlier). A larger C will result in more classification errors. We used $C = 1$ which provides normalized values for α. Those training vectors for which $\alpha > 0$ are termed support vectors.

6.2 Linear Support Vector Machines

Samples are linearly separable if it is possible to partition the space between two different classes of samples using straight lines as shown in Fig. 6.3. For the linearly separable case, the support vector algorithm simply looks for the separating hyperplane with the largest margin as shown in Fig. 6.1.

6.2.1 Non-linear Support Vector Machines

Samples are non-linearly separable if two different classes of samples separable, but it is not possible to partition them using straight lines as shown in Fig. 6.4. In the case where the decision function is not a linear function of the sample data, a kernel trick can be used to accomplish this in an astonishingly straightforward way. For the non-linearly separable case, map the training data nonlinearly into a higher-dimensional feature space via a mapping function ϕ, and construct a separating hyperplane with maximum margin there. This yields a nonlinear decision function in input space. By the use of a kernel function, it is possible to compute the separating hyperplane without explicitly carrying out the map into the feature space

6.3 Kernels

The use of kernel functions provides a powerful and principled way of detecting nonlinear relations using well-understood linear algorithms in an appropriate feature space. The Kernel maps the data into some other dot product space F called the feature space via a nonlinear map,

$$\Phi : R^N \to F, \tag{6.6}$$

and perform the linear algorithm in F. This only requires the evaluation of dot products.

$$K(x, y) := (\phi(x) \cdot \phi(y)) \tag{6.7}$$

If F is high-dimensional, the evaluation of dot products will be very expensive. There is a computational shortcut which makes it possible to represent linear patterns efficiently in high-dimensional spaces to ensure adequate representational power. The shortcut is what we call a kernel function. In some cases, we can use simple kernels that can be evaluated efficiently. For example, the polynomial kernel

$$K(x, y) = (x \cdot y)^d \tag{6.8}$$

can be shown to correspond to a map into the space spanned by all products of exactly d dimensions of for d=2 and , for example, we have

$$(x, y)^2 = ((x_1, y_1) \cdot (y_1, y_2))^2 = ((x_1^2, \sqrt{2}x_1x_2, x_2^2) \cdot (y_1^2, \sqrt{2}y_1y_2, y_2^2))^2 \tag{6.9}$$

$=(\phi(x) \cdot \phi(y))$, defining $\phi(x) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$.
For every kernel we can construct a map ϕ such that Equation (6.7) holds. Radial Basis Function (RBF) kernels and sigmoid kernels are defined as follows:

$$K(x, y) = exp(\frac{-||x - y||^2}{2\sigma^2}) \tag{6.10}$$

Sigmoid kernels:

$$K(x, y) = tanh(K(x, y) + \Theta) \tag{6.11}$$

In input space, the hyperplane corresponds to a nonlinear decision function, whose form is determined by the kernel (see Fig. 6.5).

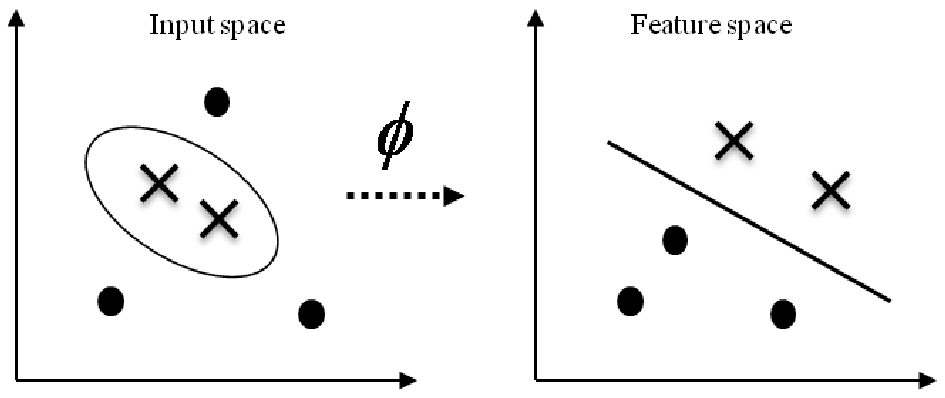

Figure 6.2: Non linear samples inseparable in the input space, rendered separable in feature space

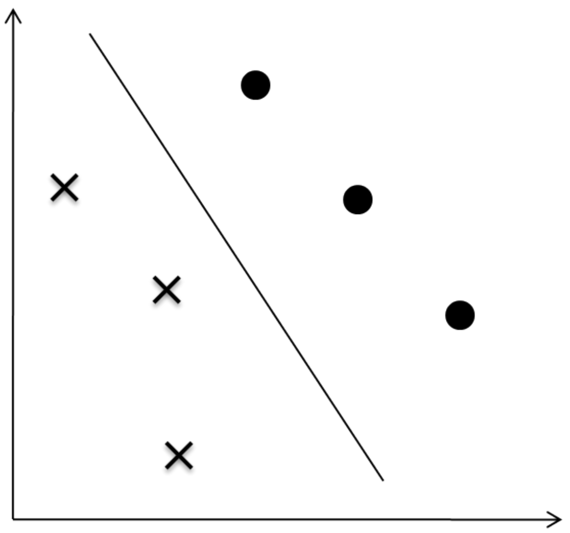

Figure 6.3: Linear samples

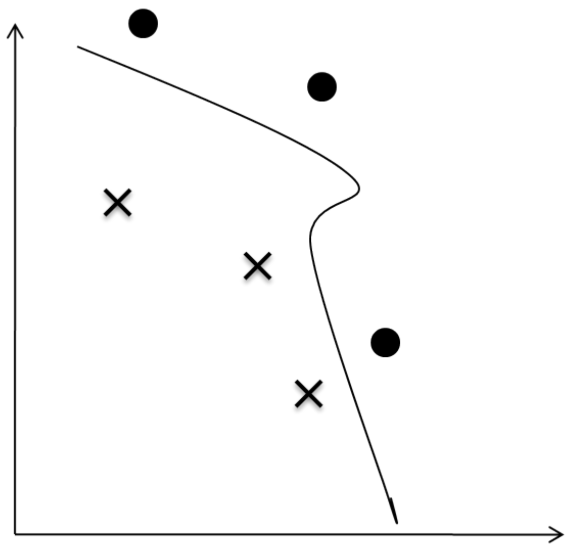

Figure 6.4: Non-linear samples

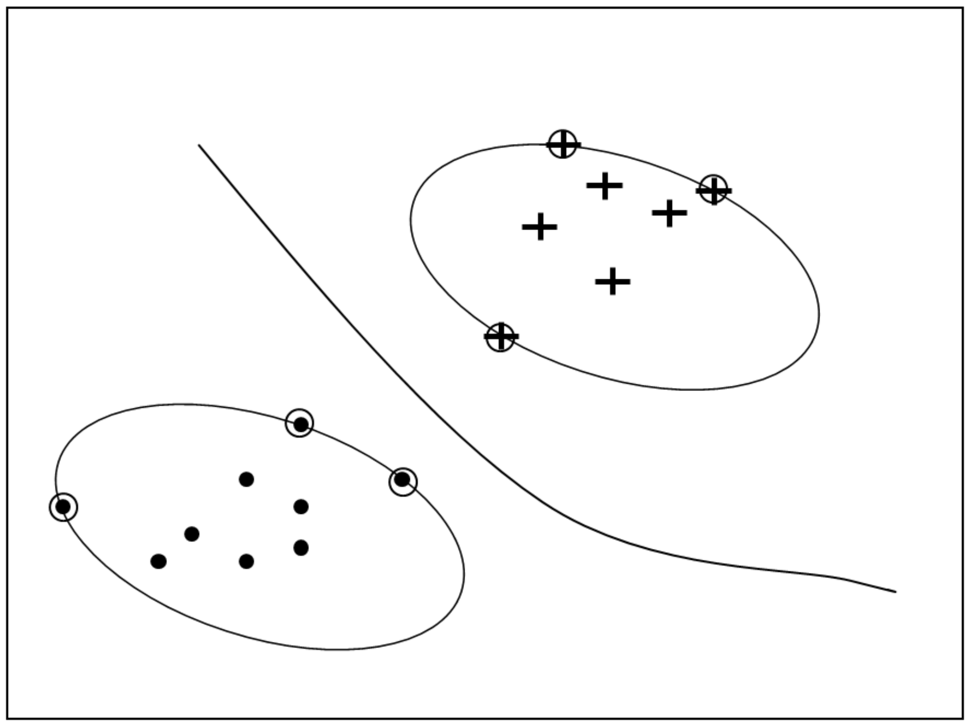

Figure 6.5: Decision boundary and support vectors when using a Gaussian kernel

Chapter 7

Support Vector Machine & Particle Swarm Optimization

In this chapter, we will introduce a Particle Swarm Optimization (PSO) method which can be employed to solve the quadratic programming problem issued when creating a support vector machine. We will also discuss a modification for preventing premature convergence, a problem commonly observed in evolutionary computation algorithms such as PSO and Genetic algorithms (GA).

7.1 Particle Swarm Optimization

Image synthesis of dynamic objects such as clouds, smoke, water, and fire are known to be difficult to simulate due to their fuzziness. Researchers have studied particle systems in which particles have their own behavior [41] and have tried to simulate such natural dynamic objects. Many scientists have also been interested in the movement of a flock of birds or a school of fish to discover underlying rules that make possible their aggregate motion. The aggregate motion enables them to find food quickly, and protects them from predators through early detection and the spread of information. Intrigued by such social behavior, Kennedy and Eberhart [31] developed the particle swarm optimization (PSO) method to apply their natural behavior to problem solving. In PSO, particles simulate the social behavior of a flock of birds, which cooperate with one another to find food (goal), in a way that each one remembers its own best location ever visited called "*local best*," and shares the local information with their neighbor to identify "*global best*" location within a group. The local information refers to individual cognition and the global information to social interaction. Using the local and global information a swarm of particles are able to cooperate and explore the solution space effectively to find an optimal solution. One of the advantages of PSO is the particle's prompt convergence on solutions by exploring the solution space in a fast speed like a flock of birds moving fast, but astonishingly perfect harmony. However, the solution by early convergence may become a local solution if done prematurely. This drawback of PSO can be contributed to its lack of capability to indicate premature convergence. The premature convergence phenomenon is commonly observed in evolutionary methods such as Genetic Algorithms (GA). In the case of a GA, it is known that high selection pressure in choosing only superior offsprings for new generations results in premature convergence. This is because the high selection pressure restricts diversity of new population, making a search for solution space limited to local areas. Therefore, a search may get stuck in a local optimum. Riget and Vesterstrlm attributed the premature convergence of PSO to the fast information flow between particles, which causes particles to cluster early around local minima [43]. Since little diversity among population is considered the main cause of premature convergence, many researchers [105] [112] [103] have tried to avoid premature convergence in a way that provides sufficient diversity for particles at the indication of premature convergence. However, it remains a very difficult problem to identify the sign of premature convergence. Furthermore, using such a reactive approach may be difficult to escape local minima for optimization problems. In this research, we propose a novel cluster-PSO (CPSO) using self organizing map (SOM), which is known as an unsupervised learning method called "*Kohonen vector*" [45]. Mohan and Al-kazemi [103], and Seo et al.[117] used clustered particles to search the solution space concurrently. However, the search may not be adaptive to the type of solution space since the population of group is static. On the other hand, CPSO using SOM technique is able to make the search adaptive to the solution space by dynamically updating the population of clustered particles. SOM, the unsupervised learning method is able to cover large solution space effectively by periodically clustering particles around *Kohonen* vectors randomly created, and thereby enables preventing particles from getting stuck in local optima.

7.1.1 Background

The PSO model simulates a swarm of particles moving in a n-dimensional solution space where a particle corresponds to a candidate solution characterized by n attributes. It is represented in the solution space by its position vector $\vec{x}_i$, and a velocity represented by a velocity vector $\vec{v}_i$. The velocity of i^{th} particle of the swarm and its projected position in the d^{th} dimension are defined by the following two equations:

$$\vec{v}_{id} = \vec{v}_{id} + c_1 \cdot rand() \cdot (\vec{l}_{id} - \vec{x}_{id}) + c_2 \cdot rand() \cdot (\vec{g}_{id} - \vec{x}_{id}) \tag{7.1}$$

$$\vec{x}_{id} = \vec{x}_{id} + \vec{v}_{id} \tag{7.2}$$

where $i = 1, \cdots, n$ and n is the size of the swarm, $d = 1, \cdots, m$ and m is the number of dimension in the solution space. Each particle remembers its own best position in $\vec{l}_{id}$. Furthermore, the best position found by any particle i is stored in $\vec{g}_i$ to share the information with neighbors within the swarm. New velocity of particles is determined by (7.1), where c_1 and c_2 control learning rate of particles for individual cognition and social interaction, respectively, and $rand()$ is a random function in the range [0,1]. Shi and Eberhart [113] introduced a parameter *inertia weight* ω into the basic PSO:

$$\vec{v}_{id} = \omega \cdot \vec{v}_{id} + c_1 \cdot rand() \cdot (\vec{l}_{id} - \vec{x}_{id}) + c_2 \cdot rand() \cdot (\vec{g}_{id} - \vec{x}_{id}) \tag{7.3}$$

where ω determines the magnitude of the old velocity $\vec{v}_{id}$. They found the range of [0.9,1.2] a good area to choose ω from. Many researchers have tackled the premature convergence problem in PSO. They tried to overcome the premature convergence problem in a reactive way that provides diversity to particles at the indication of premature convergence so as to escape a local optimum. Krink and Riget [105] provided diversity for particles when indicating their collision, which was determined based on radius between particles, and subsequently particles bounced away. The direction in which particles should bounce away was determined randomly or by simple velocity-line bouncing in which particles continue to move in the direction of old velocity-vector with a scaled speed, resulting in U-turn. The tailored PSO outperformed the basic PSO for several benchmark functions. However, the reactive method may lead to trouble for multi-objective problems. Once converged at a local optimum, clustered on local best by their nature, particles would struggle to escape the local optimum without enormous diversity. On the other hand, CPSO explores solution space concurrently by explicitly clustering particles around sample vectors randomly created, thus being able to escape local optima for multi-objective problems. Wei et al. presented Elite Particle Swarm with Mutation (EPSM) [111]. EPSM tried to take full advantage of outstanding particles in order to avoid wasting a considerable amount of time visiting the solution space with poor fitness values. To do this particles with poor fitness is substituted by *elite particles* with better fitness. In the EPSM the diversity of particles will be decreased due to the elitism. In order to cover the little diversity they employed a mutation operator so that the global best individual may be mutated to generate a new particle. In contrast to the elitism Wang and Qiu [112] tried to give inferior particles opportunities to search solution space. Their approach was motivated by the observation that a search process is very likely to be dominated by several super particles, which may usually be turned out not really good in the long term. In order to alleviate the dominance by super particles, they introduced roulette wheel selection operator in which the selection probability of a particle was ensured to be in inverse proportion to its original fitness, and chosen a particle in the roulette wheel manner, which is expected to mitigate the high selection pressure by super particles. Their approach outperformed other published algorithms in terms of solution quality with additional computational time for fitness scaling and roulette selecting process. Veeramachaneni and Osadciw [106] claimed that particles by nature oscillate between local optima and a global optimum, wasting most time moving in the same direction to converge at a global optimum. They made particles attracted toward the best positions visited by their neighbors, and thereby particles are influenced by successful neighbors to explore all other possible solution space. This algorithm was improved by concurrent PSO algorithms [107] in which two particle groups worked concurrently, each group tracing particles independently and sharing the information for the best particle. Multi-Phase Particle Swarm Optimization algorithm employed multiple groups of particles, each changing a search direction every phase to increase population diversity [103].

7.1.2 CPSO model

The CPSO is a modified version of PSO with an additional process of clustering particles. In the CPSO model, particles are periodically clustered around sample vectors using SOM to provide particles with enough diversity to prevent them from prematurely converging to local optima.

The CPSO procedure is described in Procedure 2.

Procedure 2 The Procedure of CPSO

```
Step 1) Initialize particles
Step 2) Randomly create sample vectors (particles)$\vec{y_i}$ on the solution space, where $1 \leq i \leq k$, $k$ being the maximum number of sample vectors
Step 3) Traverse each particle $\vec{p_i}, 0 \leq i < n$, to find the best matching particle (BMP) for each sample vector by similarity between sample vector and particles using euclidean distance.
Step 4) Update the velocity of particles in the neighborhood of BMP by pulling them closer to sample vectors using the following formula:
    $V\vec{p}(t+1) = V\vec{p}(t) + \Phi(p,t)\alpha(t)(\vec{y}(t) - V\vec{p}(t))$    (7.4)
Step 5) Evolve particles using PSO.
```

In eq. 7.4, $V\vec{p}(t+1)$ is new velocity of particles, $\alpha(t)$ controls the learning rate where t is the generation number of particles and $\Phi(p,t)$ is the neighborhood function which determines the degree of neighborhood between BMP and particle p. We took a Gaussian function as a neighborhood function for particles which denotes the lateral particle interaction and the degree of excitation of the particle. The Gaussian function which returns values between 0 and 1 is commonly used a a simple model simulating a large number of random values.

Gaussian function for particles returns a value close to 1 if the particle is close to BMP (neighbors of BMP). Particles for which the Gaussian function returns values close to 1 are considered neighbors of BMP. The number of neighbors is reduced as the generation number grows. From Step 1 through Step 4, particles are clustered around sample vectors. This process enables particles to be relocated around sample vectors, thus giving them a chance to explore a new possible solution space which may contain an optimal or near optimal solution. In Step 5, each cluster of particles is evolved using PSO. These processes of clustering and evolving particles are iterated until the generation number is exhausted or a stopping condition which identifies no more improvements in fitness on objective functions is met. Fig. 7.1 shows particles (clear circles) moving toward three randomly generated sample vectors (dark circles) to form clusters.

7.1.3 Simulations & Results

To run the simulations, we implemented the PSO and CPSO using the Java programming language. In the simulations, PSO and CPSO were run for three well known objective functions namely DeJong's F2, Schaffer F6 and Rastrigin's functions (also used by Kennedy [115]). The optimization performance of PSO and CPSO were compared. The objective functions are described as follows:

$$f(x,y) = 100(x^2 - y)^2 + (1-x)^2, (-2.048 < x, y < 2.048) \tag{7.5}$$

$$f(x,y) = 0.5 + \frac{(sin^2(\sqrt{x^2 - y^2} - 0.5)}{(1 + 0.001(x^2 + y^2)^2}, (-100 \leq x, y \leq 100) \tag{7.6}$$

$$f(x) = 100 + \sum_{i=1}^{10} x_i^2 - 10cos(2\pi x_i) \tag{7.7}$$

$$f(x) = \sum_{i=1}^{100} \frac{x_i^2}{4000} - \prod_{i=1}^{100} cos(\frac{x_i}{\sqrt{i}}) + 1 \tag{7.8}$$

where $-600 \leq x_i \leq 600$ De Jong's F2 function, represented by Eq. (7.5), is a two dimen-

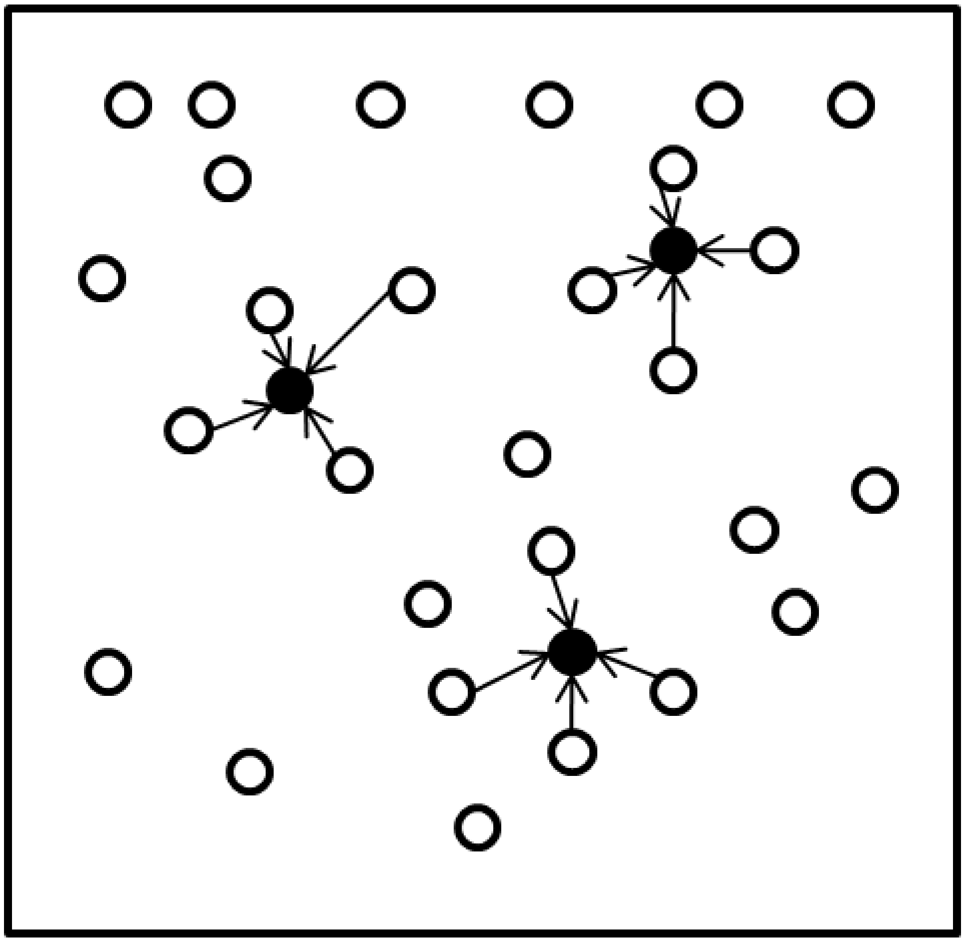

Figure 7.1: Cluster particles using SOM. Particles (circles) move toward random vectors (dark circles)

sional function with a deep valley with the shape of a parabola. The Schaffer F6 function (7.6) is known to be very difficult to optimize, having infinite local minima and one global minimum at $(x, y) = (0, 0)$. Rastrigin (7.7) and Griewank (7.8) are multimodal functions that have many local minima. Figs. 7.2 and 7.3 show Rastrigin's function and Griewank function, respectively, which have many local minima–the "valleys". Both have the global minimum at $(0 \ldots 0)$. Figs. 7.4-7.7 show minimum fitness values found by particles exploring the solution space under the objective functions described above. Fig. 7.4 shows the results of PSO and CPSO on the F2 function. For the F2 function, both CPSO and PSO performs well, early finding a minimum. Both converge early to a minimum, but CPSO continues to search a better solution (which in this case does not exist). In Fig. 7.5, it is shown that PSO

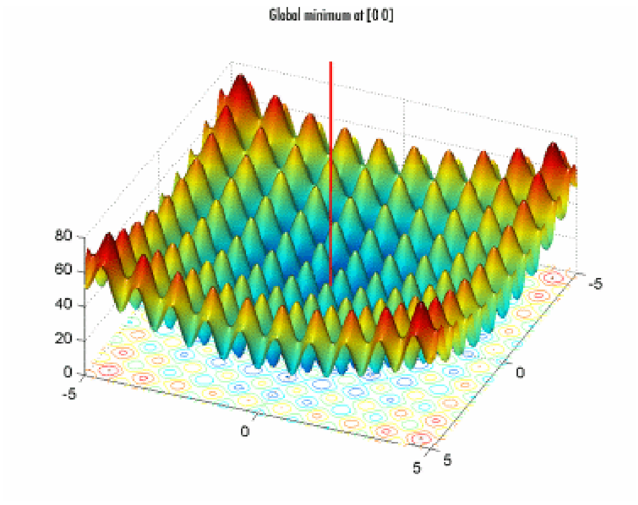

Figure 7.2: Rastrigin's Function

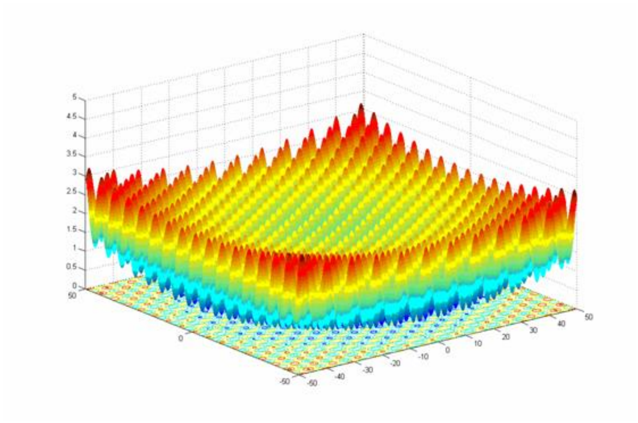

Figure 7.3: Griewank Function

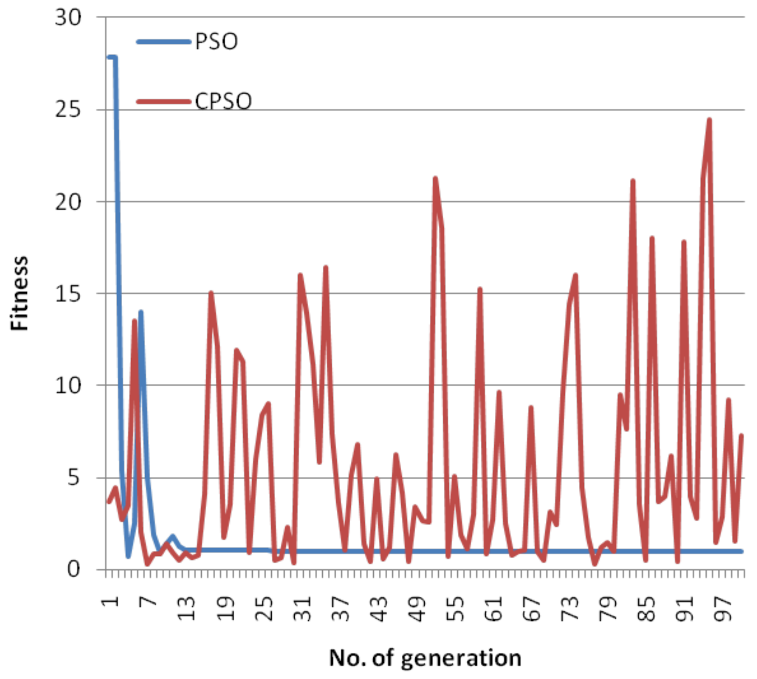

Figure 7.4: CPSO Vs PSO for De Jong's F2

prematurely converges to a solution, whereas CPSO escapes several local optima to reach global optima. Again, Fig. 7.6 shows that for the Rastrigin's optimization problem (7.7), CPSO enables particles to find a global solution, oscillating between local minima and global minimum, whereas particles of PSO converge at a local minimum. Finally, Fig. 7.7 shows CPSO outperforms PSO dramatically for very complex multimodal optimization problem. PSO early converges at local minima, whereas CPSO quickly finds a global minimum. The results clearly demonstrate that our approach is very effective for highly complex multimodal optimization problems.

7.1.4 Conclusions

We addressed the problem of premature convergence observed in PSO. In this research, we focused on providing enough diversity for particles to escape local minima. CPSO explicitly clusters particles around sample vectors to enable particles escape local minima,

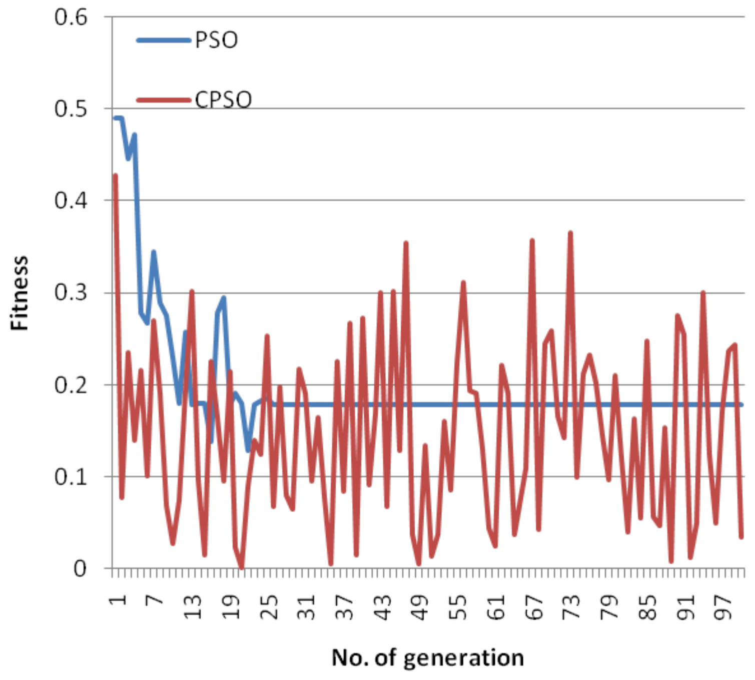

Figure 7.5: CPSO Vs PSO for Schaffer F6

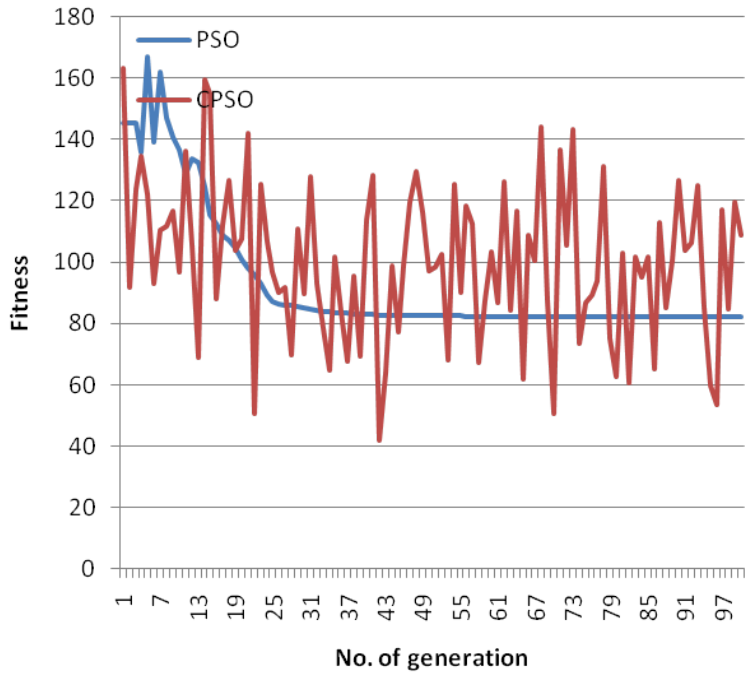

Figure 7.6: CPSO Vs PSO for Rastrigin F1

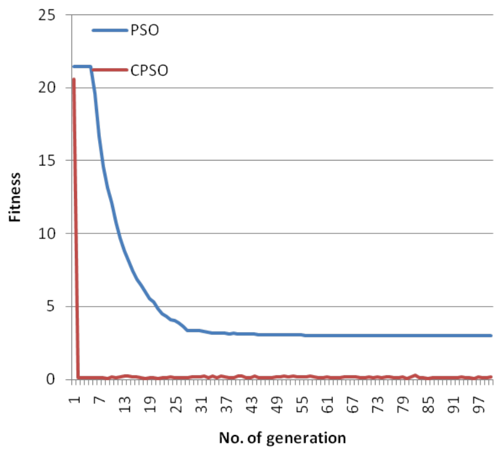

Figure 7.7: CPSO Vs PSO for Griewank

and explore new possible solution space which may contain better solutions. Simulation results show that CPSO outperforms PSO significantly for complex optimization problems and avoids local minima yielding global solutions. Although CPSO makes an extensive search within the solution space, it limits the search time by limiting the particles which explore the solution space using self organizing map. The research strongly suggests that CPSO is very effective for complex multimodal problems. In future work, we will study finding early signs of premature convergence for preventing such premature convergence.

7.2 Implementing Support Vector Machine using Particle Swarm Optimization

We implemented an SVM using PSO, namely SVM^{PSO} which is written in Java. SVM is a parameterized function whose functional form is defined from training sample data. In order to implement the SVM we have to solve a convex optimization problem on the sample training data. Solving the convex optimization problem means finding an optimal hyperplane that separates the sample training data with maximal margin. The training sample data consists of a set of N examples. Each example consists of an input vector, x_i, and a label, y_i, which describes whether the input vector is in a predefined category. There are N free parameters in an SVM, each of which corresponds to N examples. SVM will fit the function from a set of examples, that is to solve the convex optimization problem defined in eq. 6.4 and 6.5. The convex optimization problem can be solved by deciding the α values constrained to eq. 6.4. We solved the convex optimization problem using PSO by evolving the α values. The PSO procedure has been defined as:

$$\vec{v}_{t+1} = \vec{v}_t + C_1 * rand() * \vec{lbest} - \vec{x}_t) + C_2 * rand() * (\vec{gbest} - \vec{x}) \tag{7.9}$$

$$\vec{x}_{t+1} = \vec{x}_t + \vec{v}_{t+1} \tag{7.10}$$

The objective function used in PSO has been defined as:

$$\frac{1}{2} \sum_{i,j=1}^{N} \alpha_i Q_{ij} y_j - \sum_{i=1}^{N} \alpha_i \tag{7.11}$$

The pseudo code for solving the convex optimization problem for SVM^{pso} is described in procedure 3. In order to construct an SVM (step 1) needs a matrix Q where Q is an N x N

Procedure 3 The procedure of SVM^{pso}

```
  (1) Construct matrix $Q$
  (2) Initialize particles representing $\alpha$
  **while** until stopping condition is met **do**
    **for** each particle **do**
      (3) Calculate fitness value using the objective function (7.11)
      (4) If the fitness value is better than the best fitness value (pBest) in history set
      current value as the new pBest
      (5) Choose the particle with the best fitness value of all the particles as the gBest
      (6) Calculate particle velocity according to equation (7.9)
      (7) Update particle position according to equation (7.10)
    **end for**
  **end while**
```

matrix that depends on training inputs x_i, and labels y_i. Furthermore, Q_{ij} refers to i^{th} row and j^{th} column in the matrix and its value is computed by dot product between (x_i and x_j). In step (2) we generate particles representing α values, each of which is multi-dimensional vector, and initialize them. The α values are evolved iteratively through steps(3)-(7) until stopping condition is met.

7.3 Application of SVM

As a preliminary study we applied the SVM^{pso} to the problem of categorizing text documents, and compared its performance against SVM^{light} written in C by Joachims [4]. Categorizing text documents means classifying text documents into predefined classes by topic. In categorizing text documents, the documents should first be represented in vector form based on the frequency of unique words within a document. The simplest representation most commonly used to assess the topic of a document is known as the vector space model (VSM). The simplest possible version of the VSM is the representation of a document as a bag-of-words. In this model, a text document can be represented in the multi-dimensional space as a vector which consists of the weight of unique words within the document. For the training of SVM, text documents are to be first classified manually, and have a corresponding label. The labeled documents are used to train the SVM. In order to construct an SVM we have to find an optimal hyperplane that separates two different classes of documents with maximal margin. The optimal hyperplane is perpendicular to the two hyperplanes that connect the documents with maximal margin. These two hyperplanes and training instances, lying on each hyperplane, called “support vectors”, serve as a decision model, namely support vector (SV) model. Any decision model only consists of support vectors. In this regard, we can say that support vectors are a reduced set of training instances. They can also be seen as most informative among training instances in terms of information retrieval. Therefore, whether unknown documents belong to a predefined category or not is determined by these support vectors. We trained SVM^{pso} with 2000 samples, 1000 positive and 1000 negative samples and tested with 600 samples, 300 positive and 300 negative, for four different kernels. The experiment results are shown in Fig. 7.8. From the results, it is observed that the performance of SVM^{pso} is very close to SVM^{light} except with the RBF kernel. The low performance found in SVM^{pso} against SVM^{light} for RBF kernel may be contributed to parameters not properly set to control the RBF kernel.

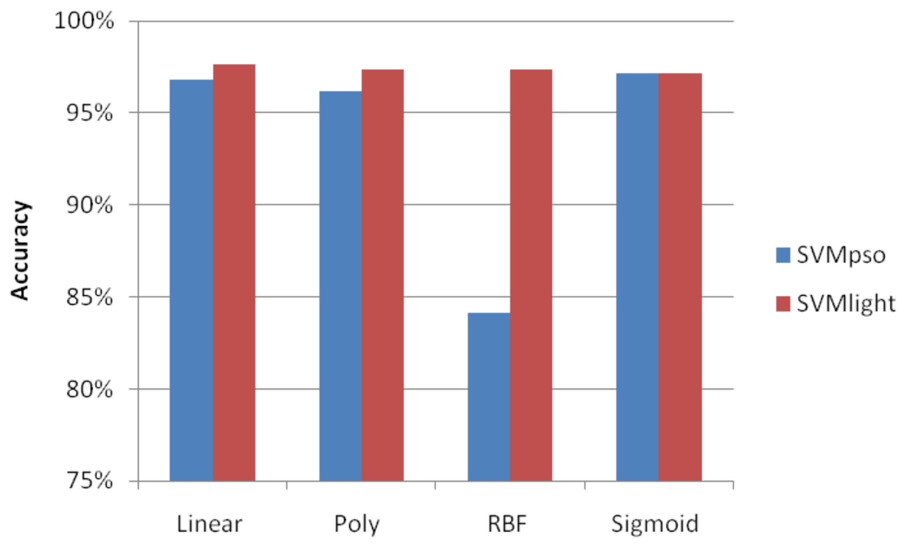

Figure 7.8: SVM^{pso} Vs SVM^{light} by kernel

Chapter 8

Support Vector Scheduler (SVS) framework

The SVS meta-scheduler has been illustrated in Fig. 8.1. It shows m target machines. The SVM is trained using sample data to create a SV model. Once trained, the SV model selects one of the two heuristics: *MinMin* or *MaxMin* to perform the scheduling. The input to the SVS is a batch of tasks, whose characteristic can be represented by an ETC (Expected Time to Compute) matrix as shown in Table 8.1. The ETC matrix indicates the estimated expected execution time of tasks on different machines. The SVS operates in three phases to make a heuristic selection:

- Phase 1: Generation of sample training data.
- Phase 2: Construction of a SV model.
- Phase 3: Heuristic selection.

In Phase 1, sample training data x_i are generated randomly. In our experiments we used uniform as well as Gaussian distribution function for random generation of data. The generation of training data has been discussed in detail in Section 4.1. Samples are assigned labels y_i based on the simulation results obtained upon running the *MinMin* and *MaxMin* heuristics on the samples. If *MaxMin* produces lower makespan than *MinMin*, the sample is labeled '+' otherwise it is labeled '-'. In Phase 2, we train the SVM using the training samples to generate an SV model. In Phase 3, the SVS makes a heuristic selection using the SV model. It evaluates the properties of the batch of tasks, accordingly chooses either the *MinMin* or *MaxMin* heuristic for scheduling. The selected heuristic eventually maps all tasks within the batch onto machines. The phases of the SVS meta-scheduler have been described in detail next.

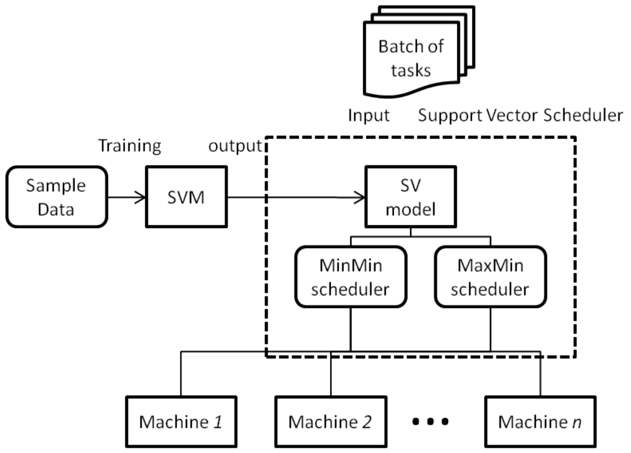

Figure 8.1: Support Vector Scheduler (SVS) Framework

8.1 Phase 1: Creating sample training data

The format of a single sample training data has been shown in Fig. 8.2. It starts with a label field whose value could be plus (+) or minus (-), and is followed by task heterogeneity within the batch, machine heterogeneity, and machine ready times, respectively. The machine ready times are not directly used by the SVM, rather they are used by *MinMin* and *MaxMin*. The labels '+' and '-' represent the *MaxMin* and *MinMin* classes respectively. The binary SVM classifies each input task into either of these two classes. Other fields of the training data represent the statistical information of the tasks within the batch and the machines. These fields represent the minimum, maximum, average and standard deviation of task and machine heterogeneity.

In order to generate training data in the format of Fig. 8.2, we first generate the ETC matrix randomly. A sample ETC matrix for a batch of ten tasks has been shown in Table 8.1. It represents the characteristics of tasks and machines that can be used to execute these tasks. It lists tasks and their expected execution times on four different machines. The expected execution times have been normalized within 1..999. The range is wide enough so that it can represent sufficient heterogeneity. The variances of each row σ_i and each column σ_j of ETC matrix were computed. The σ_i values represent machine heterogeneity whereas σ_j values represent task heterogeneity. The task heterogeneity field values for the input data

Table 8.1: ETC for a batch of tasks

Batch task	Machine			
	m_0	m_1	m_2	m_3
t_0	351	972	303	45
t_1	948	892	891	767
t_2	915	384	228	499
t_3	915	166	36	597
t_4	508	244	654	879
t_5	836	27	935	872
t_6	334	117	954	307
t_7	86	535	993	195
t_8	609	745	161	866
t_9	126	701	295	782

(Fig. 8.2) are then avg (σ_i), std (σ_i), min (σ_i) and max (σ_i). The machine heterogeneity values for the input data (Fig. 8.2) will then be: avg (σ_j), std (σ_j), min (σ_j) and max (σ_j). The machine ready times, $R_j$were randomly generated. Next, the label field was generated. To do so, *MinMin* and *MaxMin* were run for each task. The *MinMin* and *MaxMin* heuristics used the ETC matrix to compute the completion time of each task on different machines. The completion time of task i on machine j can be defined as follows:

$$C_{ij} = E_{ij} + R_j \tag{8.1}$$

where R_j is the ready time of machine j. The *MinMin* and *MaxMin*, after each assignment, update the ready times. The ready times were also normalized. The training data were labeled by classifying each sample into either of the two classes. It was labeled *MinMin* class if its total completion time is shorter using the *MinMin* heuristic than the *MaxMin* heuristic. Finally, we balance the number of samples such that half of the samples belongs to *MinMin* class and another half to the *MaxMin* class. Using these balanced samples, we train the SVM to create an SV model which is used later in SVS.

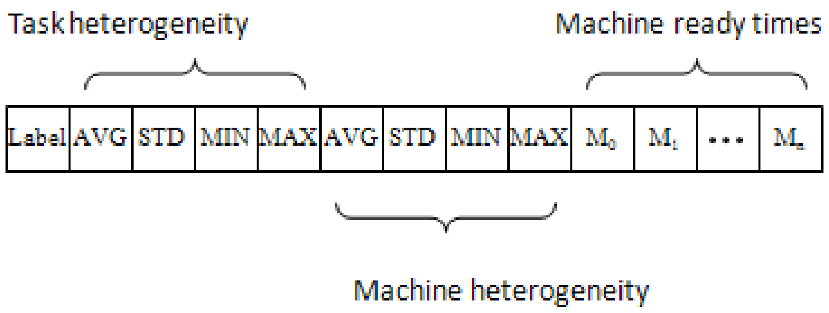

Figure 8.2: The format of the input data

8.2 Phase 2: Constructing a SV model

Creating an SV model consists of a process of finding the optimal hyperplane that separates the sample training data into two distinct regions by a maximal margin. The samples, as represented in the vector form on the multi-dimensional input space, are not linearly separable. Therefore, they are transformed via a mapping function ϕ from the non-linear input space into a higher dimensional linear feature space F where they are easily separable. The mapping function ϕ can be described mathematically as:

$$\phi : x \to \phi(x) \in F \tag{8.2}$$

It turns out, it is not necessary to explicitly compute ϕ to conduct the mapping. A kernel function is a computational shortcut that can help achieve the mapping [108]. In general, such a kernel function k can be mathematically described as follows.

$$k(x, z) = \langle \phi(x) \cdot \phi(z) \rangle \tag{8.3}$$

where $x, z \in \mathbb{R}^N$ represent input vectors and $\langle \cdot \rangle$ represents their inner product. Usually z is a reference point; in our case it represents a support vector, later represented by the symbol X_i. In our simulations, we used the widely employed Radial Basis Function (RBF) kernel, which is a Gaussian kernel and is represented as:

$$K(x,z) = exp(\frac{-||x-z||^2}{2\sigma^2}) \tag{8.4}$$

where σ controls the flexibility of the kernel. We can find the optimal hyperplane by solving the convex optimization problem (6.4) for α. The convex optimization problem is parameterized with respect to α. The overall complexity of computing the vector α is $O(l^3)$ [108] where l is the number of samples, since solving for α involves evaluating the inner products of K and solving l linear equations with l unknowns. We solved the convex optimization problem using particle swarm optimization [31], thus finding the α values denoting the support vectors. Finally loading the support vectors in the meta-scheduler, the SV model was constructed. We note, that since the actual number of samples that were used is small (we shall see later that $l = 500$ is sufficient), the actual training phase is quick.

8.3 Phase 3: Heuristic selection

The procedure of heuristic selection in the SVS has been described in Fig. 8.4. It consists of computing the heterogeneity of the machines and heterogeneity of the tasks within the batch and the machines' normalized ready times. Based on this computation, an input vector in the format of Fig. 8.2 is created. Next, the SV model evaluates the input vector. The process of evaluating an input vector is described in Fig. 8.5. The input vector x and the support vectors $X = X_1, X_2, \cdots, X_s$ are nonlinearly mapped by mapping function ϕ, into a feature space F, where dot products $\phi(x) \cdot \phi(X_i)$ are computed. As shown in the figure, support vectors X_1 and X_2 belong to the dot ($\odot$) class whereas X_{s-1} and X_s belong to the cross ($\otimes$) class. The steps of mapping and dot products above were performed via a kernel function in a single step as $\phi(x) \cdot \phi(X_i) = K(X_i, x)$. As stated earlier, the weights $\alpha_1 \cdots \alpha_n$ were computed using Particle Swarm Optimization. Next, the output was computed as the sign ('+' or '-') of the sum of weighted products. This process is in effect tantamount to the

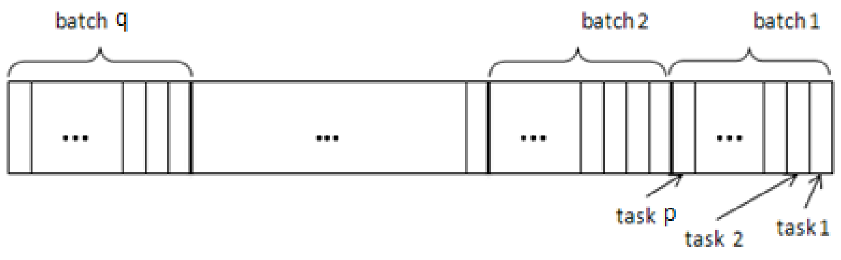

Figure 8.3: Several batches of n tasks

```
for each batch of tasks do
    1. Compute machine heterogeneity
    2. Compute task heterogeneity
    3. Generate machine ready times
    4. Create input vector $x$
    5. Evaluate $x$ and select heuristic
end for
```

Figure 8.4: The procedure of heuristic selection

input vector being compared against the support vectors for classification. Finally, the SVS makes a heuristic selection based on the classification. Consequently, the selected heuristic schedules tasks onto machines.

Evaluation of a new input vector costs $O(spq)$, where s is the number of support vectors, p is the number of tasks within a batch, and q is the number of batches as shown in Fig. 8.3. Since the number of tasks $n = p * q$ and s is constant, the complexity is $O(n)$.

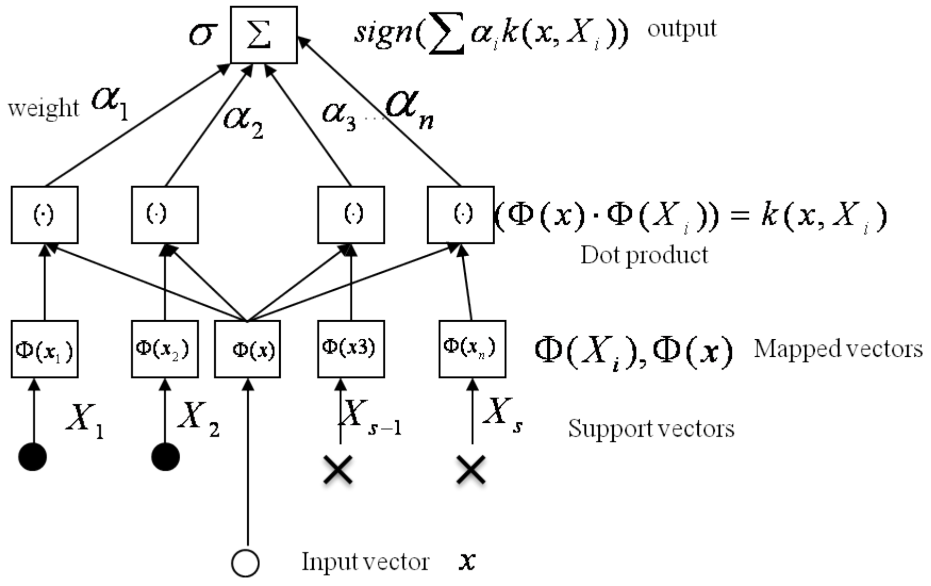

Figure 8.5: Evaluation of an input vector x by the SV model

Chapter 9

Simulation Procedure

To run the simulations, we implemented the SVS using the Java language. SVS includes the implementation of SVM using particle swarm optimization and the two batch mode heuristics, *MinMin* and *MaxMin*. In addition we simulated the performance of *Random* which randomly chooses either the *MinMin* or the *MaxMin* heuristic. The input to SVS consists of a batch of tasks that arrive following the last batch which was scheduled. In the simulation, we measure the total completion times using *MinMin*, *MaxMin*, *Random*, and *SVS*.

Using an appropriate sample size (no. of samples) for training is important in machine learning. Therefore, we conducted experiments to determine the appropriate sample size. The results of the experiments have been represented graphically in Fig. 9.1. The figure shows the average improvement in makespan produced by SVS over *MinMin* and *MaxMin* for different number of samples. From Fig. 9.1, we observe that a sample size of 500 is appropriate in our problem domain, which is the size we used for our simulations. The input parameters that were used to create test data for the simulation and their ranges are given below:

- $l = \{500\}$, the number of samples used to construct a SVM.
- $p = \{10, 20, 50, 100\}$, the number of tasks within batch.
- $m = \{4, 8, 16, 32, 64, 128\}$, the number of machines.
- $q = \{100, 500, 1000, 2000, 5000, 10000\}$, the number of batches.
- $\beta = \{0.1, 0.2, 0.3, 0.4, 0.5\}$, the heterogeneity of the negative samples.
- $\gamma = \{0.1, 0.2, 0.3, 0.4, 0.5\}$, the heterogeneity of the positive samples.

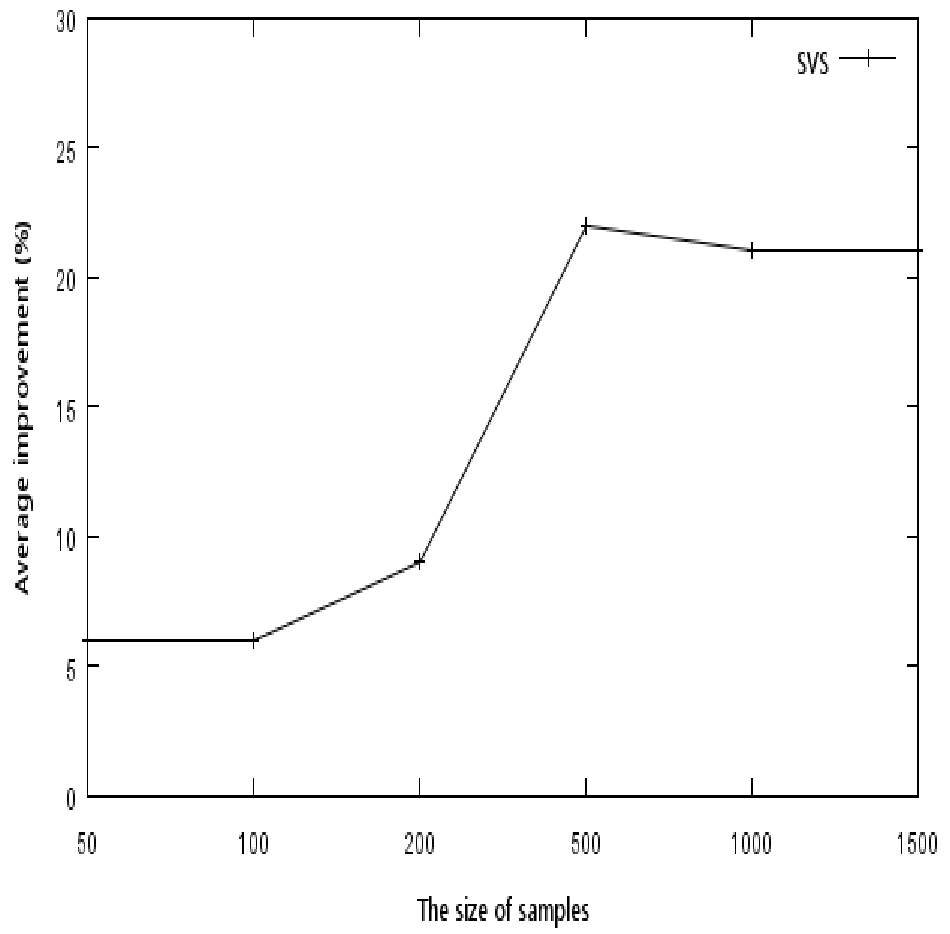

Figure 9.1: Average improvement in makespan from SVS

All possible combinations of parameters result in $4 \times 6 \times 6 \times 5 \times 5 = 3600$ test sets. For each test set, we created balanced input data as far as possible. All the simulations were independently repeated 20 times, and their results were averaged. Thus, the results in the next section summarize 3600 x 20 x 4 = 288000 data points for all four (4) of *MinMin*, *MaxMin*, *SVS* and *Random* heuristics. In order to make a validation for the performance of SVS we performed simulations for varied combinations of random number generator: Uni-Uni, Uni-Norm, Norm-Uni, and Norm-Norm. Uni-Uni combination has both training data and test data uniformly distributed and Uni-Norm combination has training data normally generated and test data uniformly distributed. Norm-Uni combination has the training data normally distributed and test data uniformly distributed. Norm-Norm combination has both training and test data normally distributed. The simulations for the validation result in $3600 \times 4 = 14400$ test sets, and data points were averaged to describe the average improvement over *MinMin* and *MaxMin*.

Chapter 10

Results and Discussion

In this chapter, the results of a preliminary study are shown and followed by the results of the simulation performed with the number of samples held constant at 500. Fig. 9.1 shows the average makespan by sample size. The average improvement goes up with the increasing number of sample, having a peak at 500, and afterward it drops slightly. In terms of probability, SVM created from large number of samples may lead to more accuracy in classification than the SVM from small number of samples. From the results it can be noted that SVM is able to improve its classification capability by creating its model from the appropriate number of samples considering problem domain. In Figs. 10.1-10.17 the performances of SVS are shown using the average improvement over *MinMin* and *MaxMin* heuristics based on the normalized average makespan. Figs. 10.1-10.4 display improvements for four different batch task sizes with respect to machine heterogeneity. It is observed in common that the improvements increase with the increasing amount of machine heterogeneity. In addition, it is noticeable that the improvement rate increases as the batch task size increases. The improvements can be attributed to the increased performance differences between two heuristics, thus resulting in a more accurate SVM with larger margin with which samples are separated. Figs. 10.13-10.16 show average improvements with respect to task heterogeneity.

It is very similar to the improvements by machine heterogeneity except there is no improvement at the point where task heterogeneity is given 30%. The small improvement may be due to imbalanced samples, causing the SV model to have a bias toward either class. Figs. 10.5-10.12 show improvements for the number of processor and task, respectively. The improvement has a peak at the number of 16 processors, and afterward it decreases.

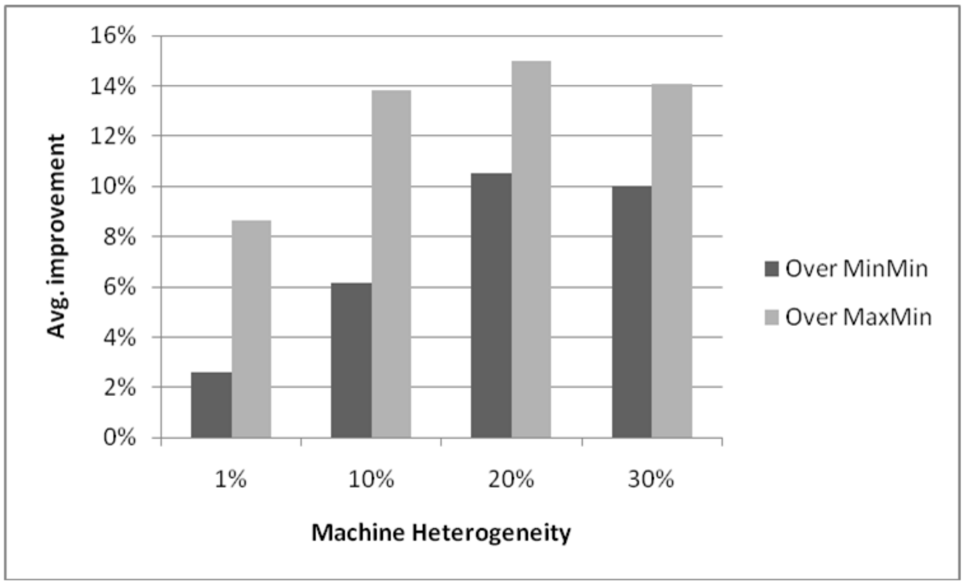

Figure 10.1: Average improvement by machine heterogeneity in batch size 10

It is shown that there are no improvements at some points in Figs. 10.5-10.8. No improvements found above for batch size 10 may be the samples not being properly created as the *MinMin* heuristic outperforms the *MaxMin* heuristic overwhelmingly. Fig. 10.9 shows that the improvements at batch size 10 fluctuate according to the size of task. On the other hand, the improvements in the larger batch size shows stability throughout all periods as seen in Figs. 10.9-10.12. The difference may be due to batch task size. The difference in performance between two heuristics by their strategies may not be significant in the small batch task size, whereas it is distinct as batch size increases. The average improvement by batch task size is shown in Fig. 10.17. The overall average improvement increases linearly with the increasing number of batch task size. The improvement is as much as 25% on average. The results indicate that there is a link between the improvement and batch task size. The improvements can be contributed to the increased margin of performance differences by batch task size, thus maximizing the capability of SVM.

To discuss the results of the simulation in terms of capability of *SVS*, we define a parameter called *Accuracy* which represents the percentage of times the selection decisions made by SVS were found to be correct. To compare the results of SVS, we also present the *Best*

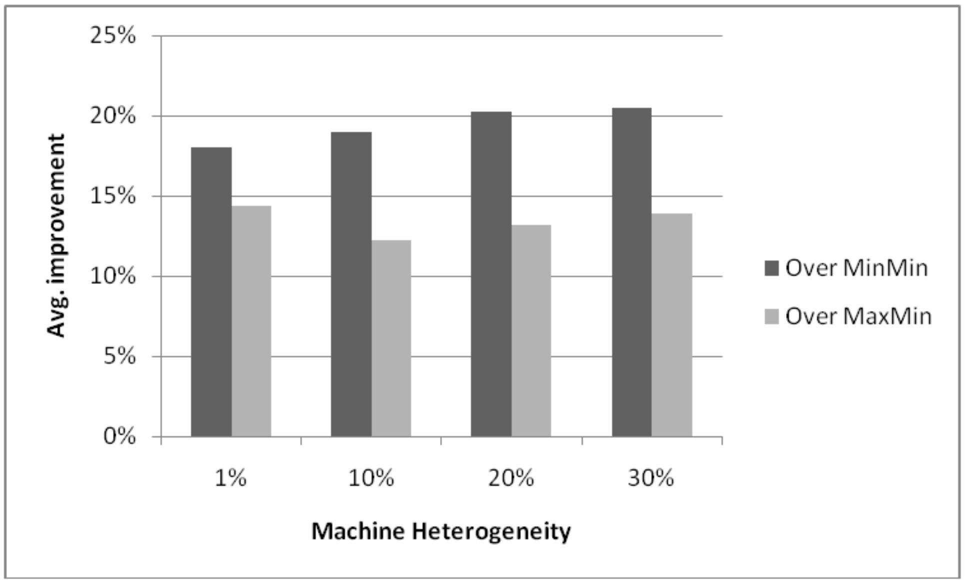

Figure 10.2: Average improvement by machine heterogeneity in batch size 20

and *Worst* makespans, where the *Best* and *Worst* refer to the makespans that can be theoretically achieved by making the best and worst selections between *MinMin* and *MaxMin*. To compare and graphically present the result in a clear fashion, the measured makespans were normalized in the range of 1..100. Since the makespans were also an average of several experiments, they are referred as normalized average makespan. The heterogeneity of tasks and machines were normalized between 1 and 5 according to the degree of heterogeneity, where 1 is lowest and 5 is highest heterogeneity. We have graphically presented the simulation results in Figs. 10.18-10.22 showing the *Accuracy* and normalized average makespans for SVS, *Best*, *Worst* and *Random* against different parameters. Figs. 10.18 and 10.19 show the average normalized makespans for different machine and task heterogeneity values. Figs. 10.20 and 10.21 show the performance of SVS for different number of machines and different number of tasks. We verified our results by independently computing *Accuracy* and makespans. From the graphs, we observe that when *Accuracy* is high, the margin between the average makespan produced by SVS and the best case is small compared to that when *Accuracy* is low. This observation further verifies the simulation data. From the graphs, it was found that the performance of SVS was very close to the best selection. We also

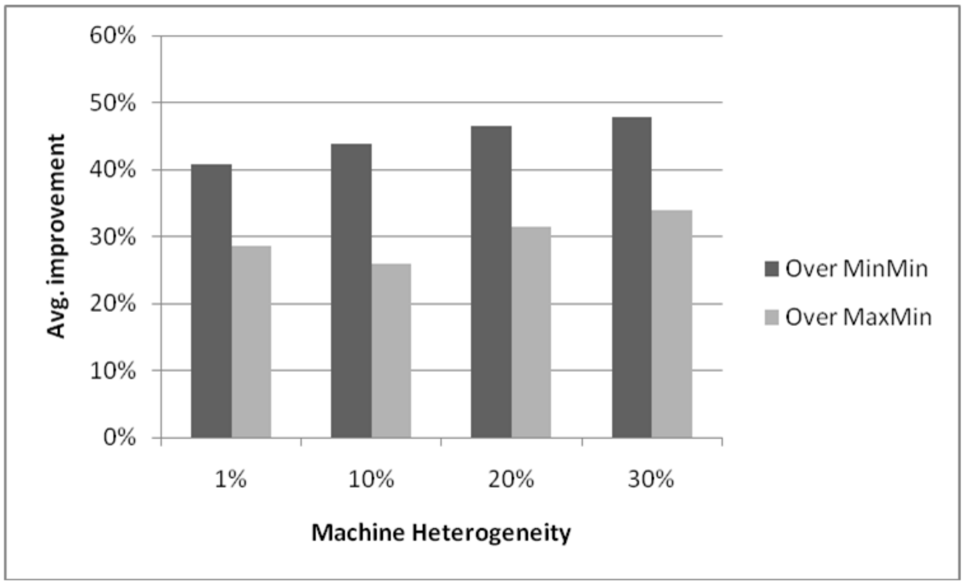

Figure 10.3: Average improvement by machine heterogeneity in batch size 50

observe that SVS achieved 46% average improvement over the worst selection and 29% over random selection. Indeed, it was within a margin 5% to the theoretical best selection. We further observe that the performance of SVS is very stable, and is close to best selection, regardless of the number of machines, number of tasks, machine heterogeneity or task heterogeneity. We found that the SVS did not need to be retrained when the heterogeneity of tasks and/or the heterogeneity of machines changes. Thus we see that SVS is adaptive to task heterogeneity and to machine heterogeneity and provides excellent performance under varying circumstances. It also scales well with no. of tasks and no. of machines.

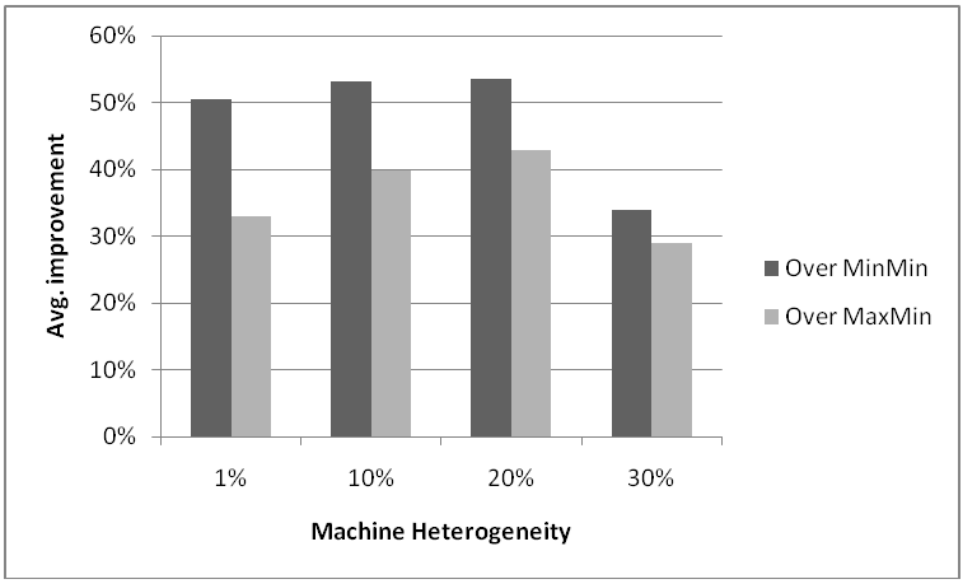

Figure 10.4: Average improvement by machine heterogeneity in batch size 100

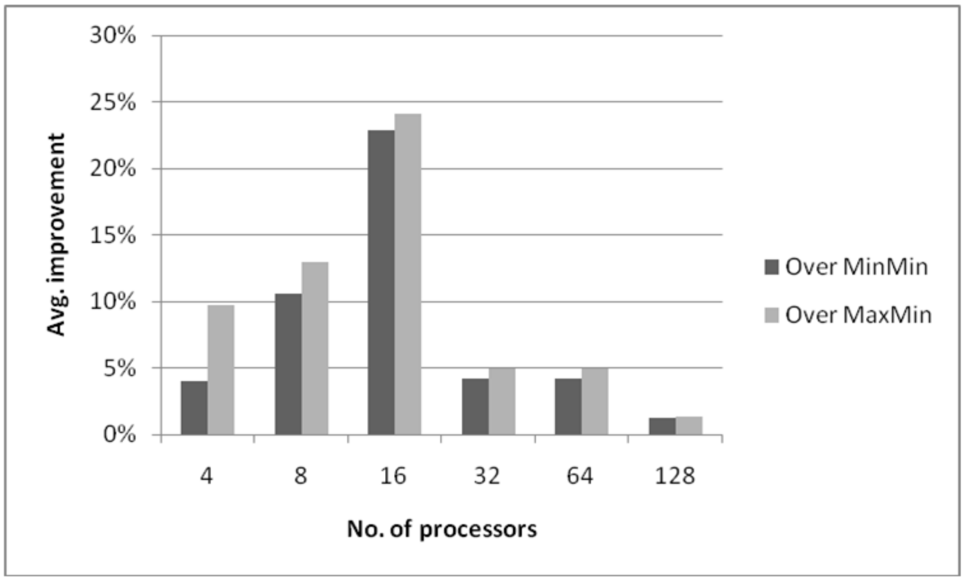

Figure 10.5: Average improvement by no. of processors in batch size 10

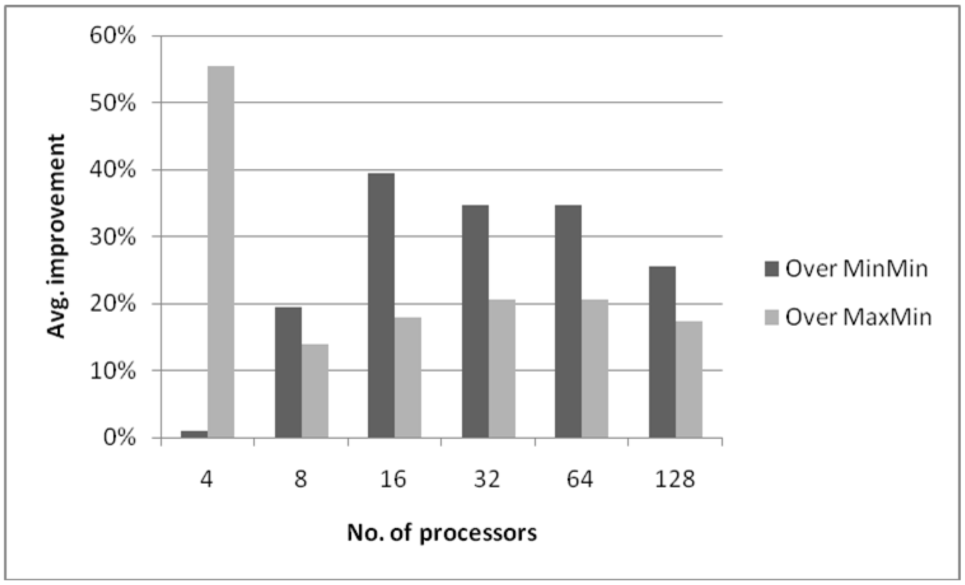

Figure 10.6: Average improvement by no. of processors in batch size 20

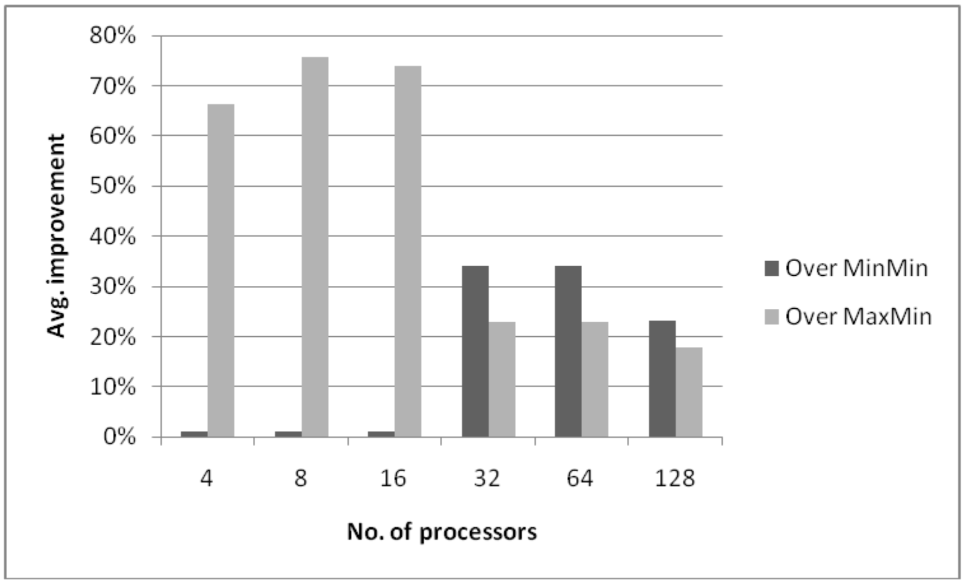

Figure 10.7: Average improvement by no. of processors in batch size 50

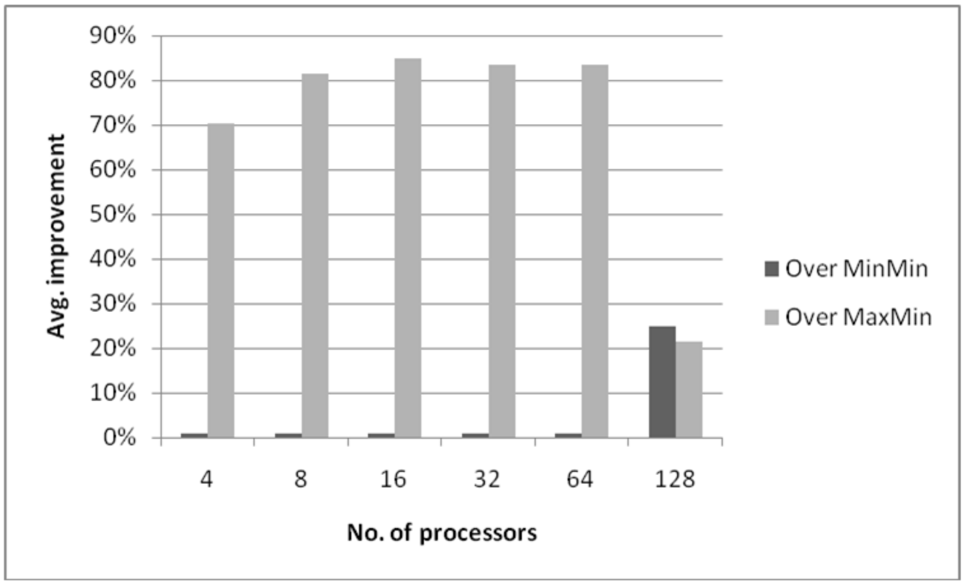

Figure 10.8: Average improvement by no. of processors in batch size 100

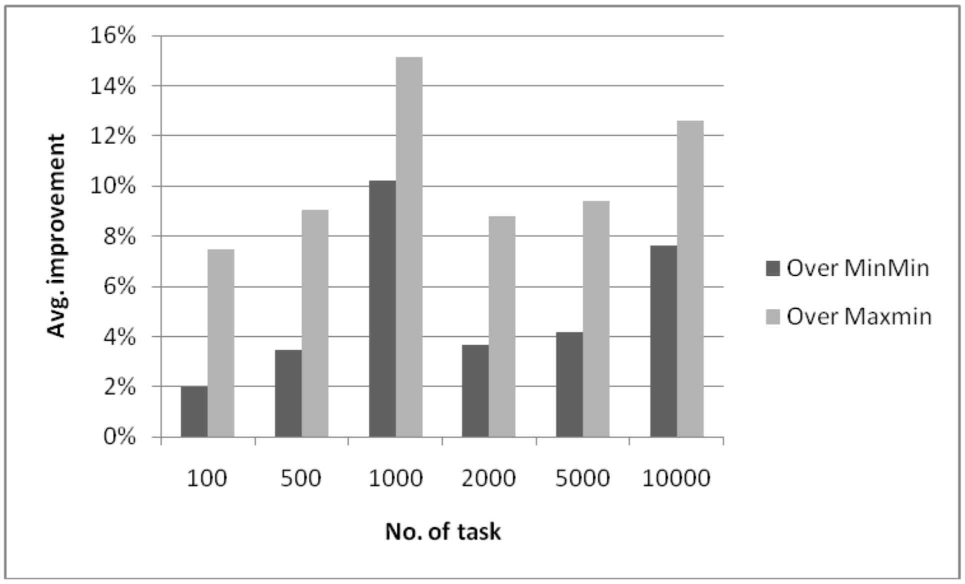

Figure 10.9: Average improvement by no. of tasks in batch size 10

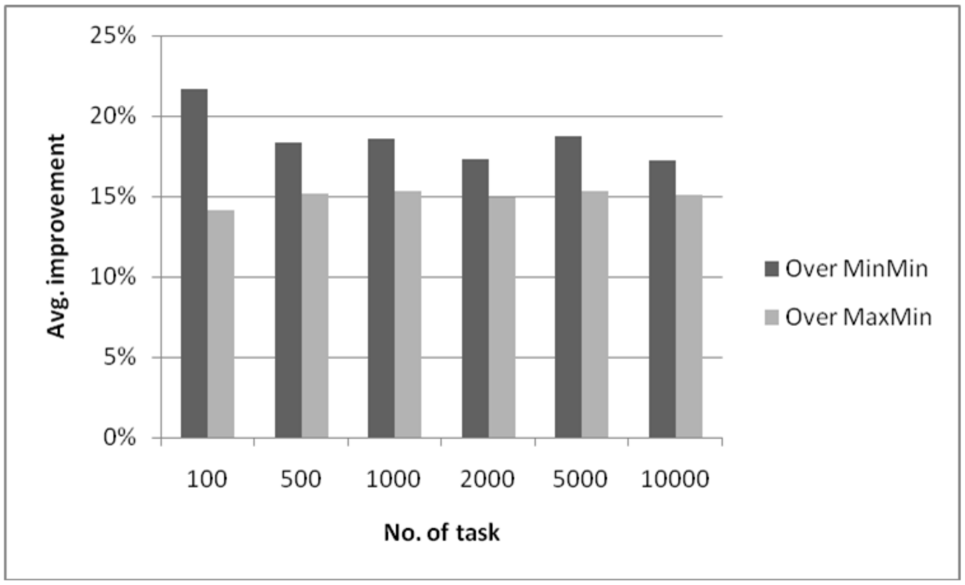

Figure 10.10: Average improvement by no. of tasks in batch size 20

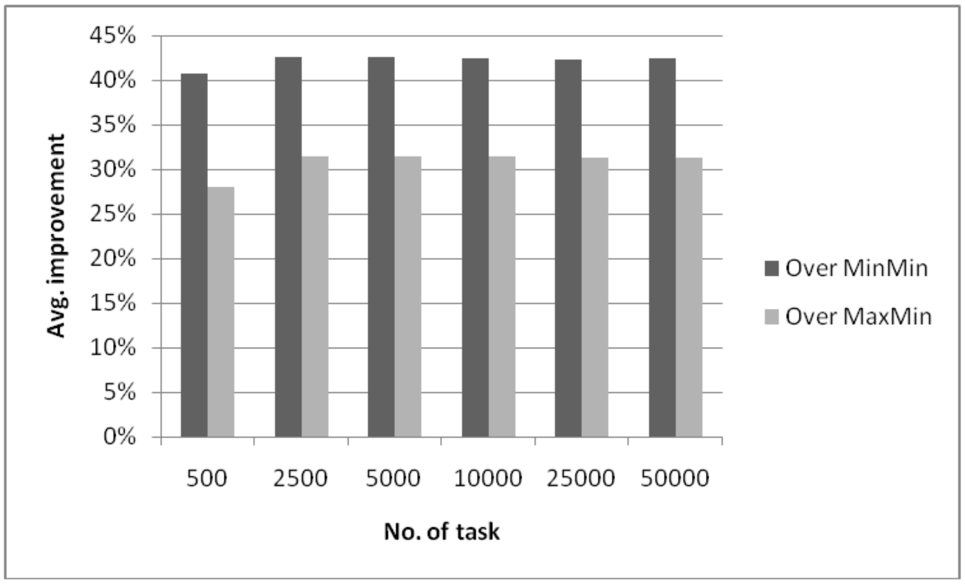

Figure 10.11: Average improvement by no. of tasks in batch size 50

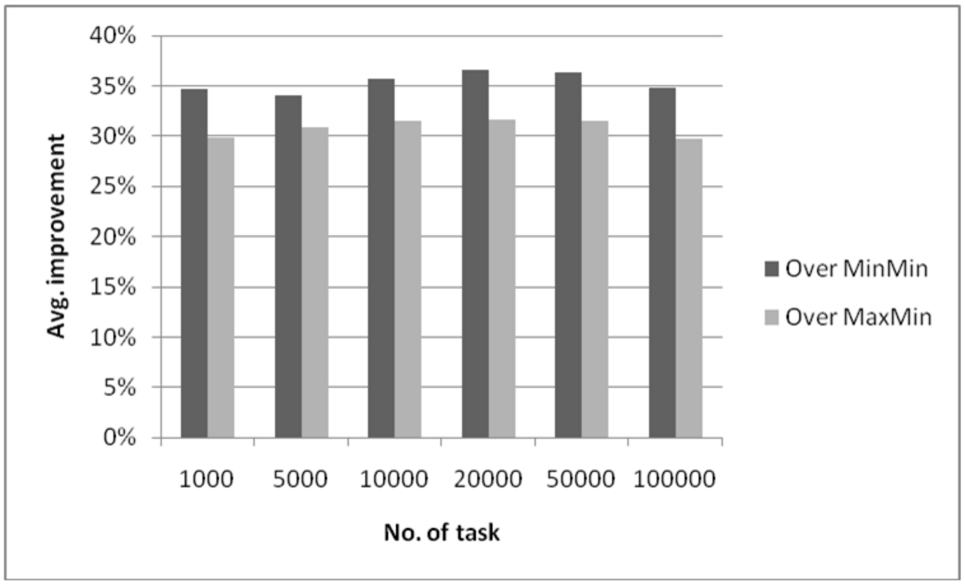

Figure 10.12: Average improvement by no. of tasks in batch size 100

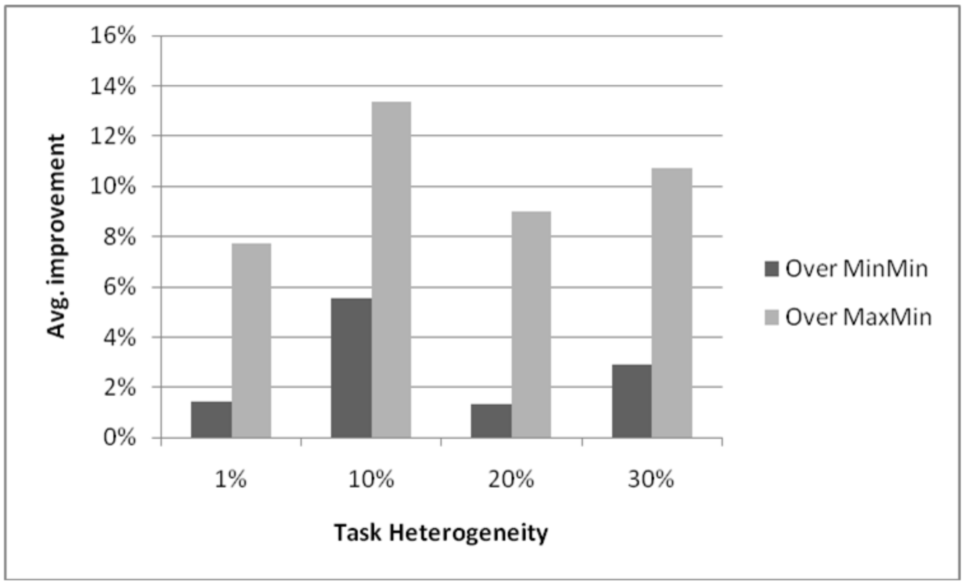

Figure 10.13: Average improvement by task heterogeneity in batch size 10

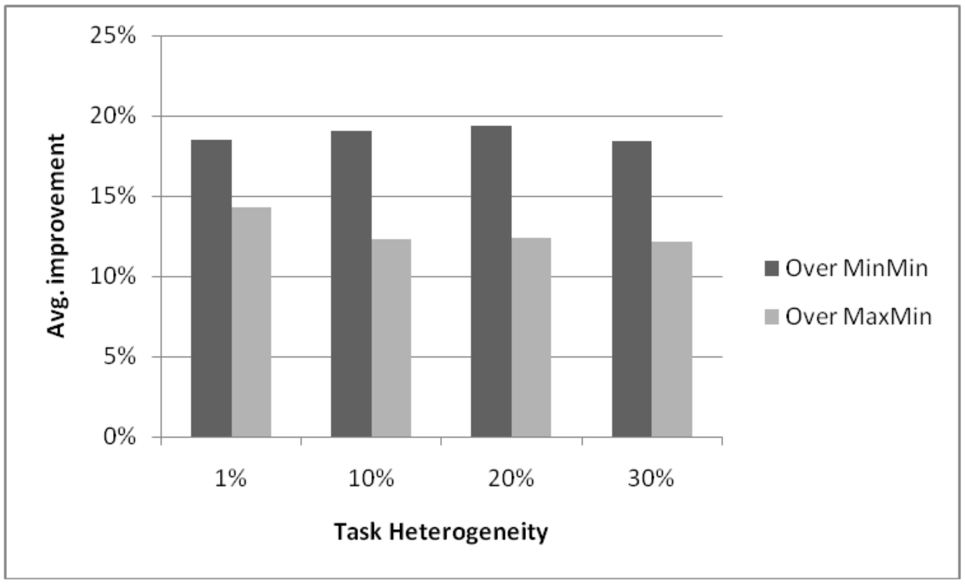

Figure 10.14: Average improvement by task heterogeneity in batch size 20

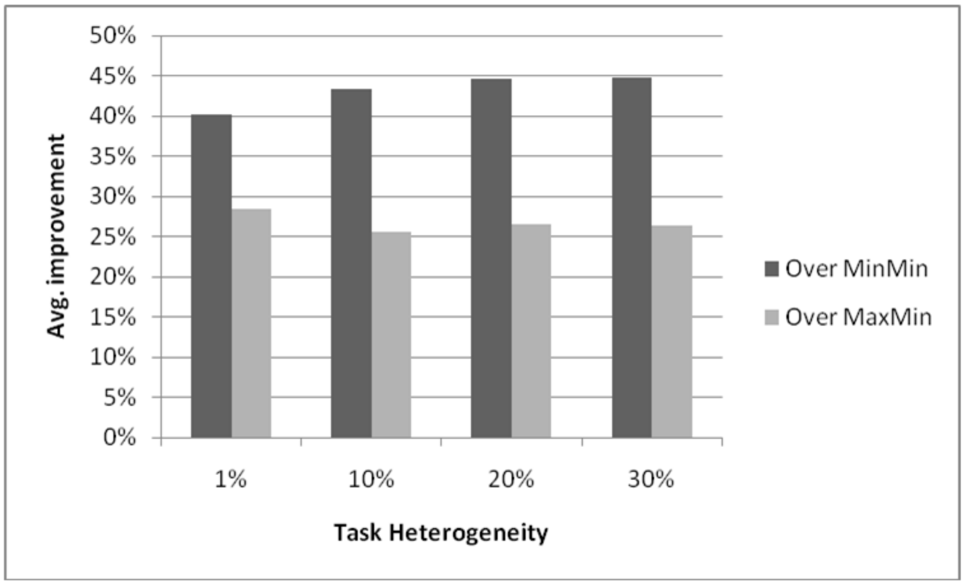

Figure 10.15: Average improvement by task heterogeneity in batch size 50

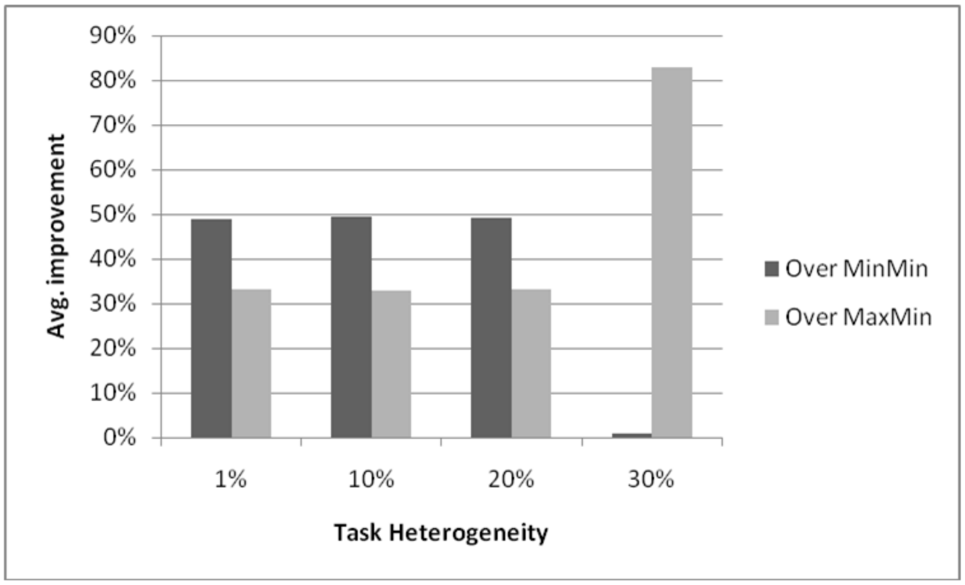

Figure 10.16: Average improvement by task heterogeneity in batch size 100

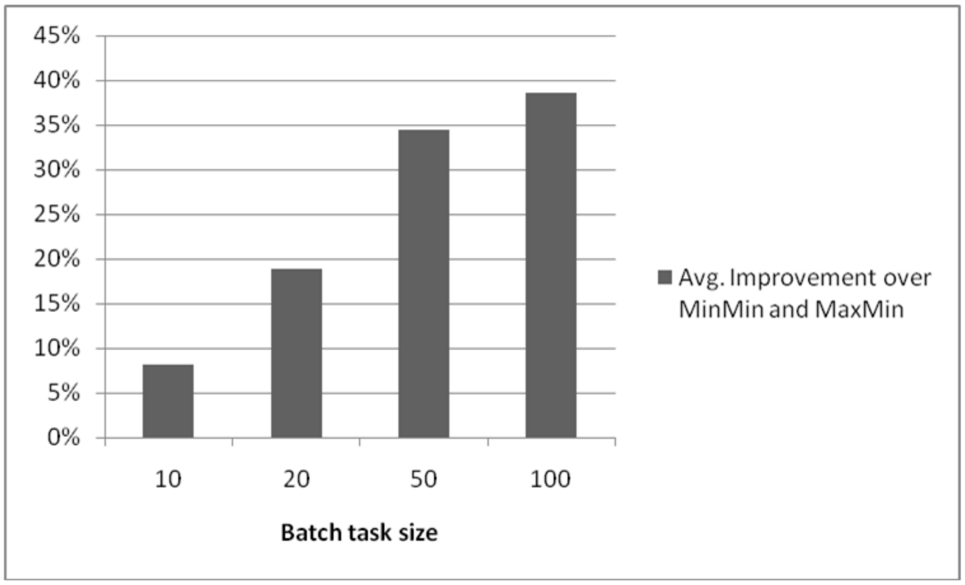

Figure 10.17: Average Improvement by batch task size

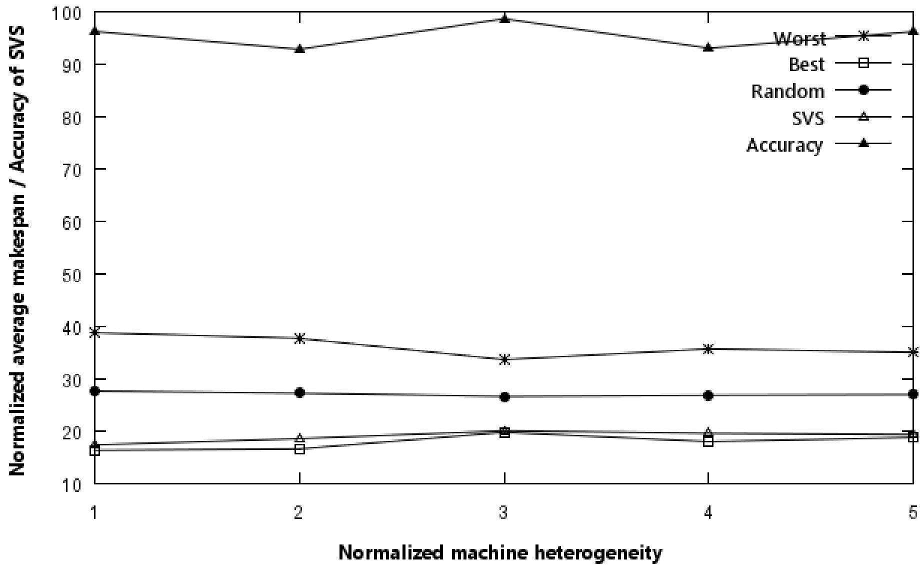

Figure 10.18: Normalized makespan and Accuracy by machine heterogeneity

Fig. 10.23 shows the accuracy of SVS by random number generator combination. It shows that Uni-Uni combination outperforms all other combinations with its accuracy close to 95%, and is followed by Uni-Norm combination. Figs. 10.24 - 10.28 show the average improvement of SVS over *MinMin* and *MaxMin* for Uni-Uni combination. From the results it can be noted that SVS has more improvement over *MaxMin* than *MinMin* in most cases. Fig. 10.27 shows SVS also achieves improvements for 16 and 128 machines although the performance margin between *MinMin* and *MaxMin* heuristics is least. Figs. 10.29 - 10.33 show the average improvement over *MinMin* and *MaxMin* for Uni-Norm combination. In the Uni-Norm combination most improvement is over *MaxMin* heuristic with low improvement over *MinMin* heuristic. The results show that regardless of no. of tasks, no. of processors, heterogeneity of tasks, heterogeneity of processors and batch sizes, SVS was able to select the best heuristic with high accuracy. Furthermore, the performance was validated using different combinations of input and training distributions functions as shown in Table 10.1.

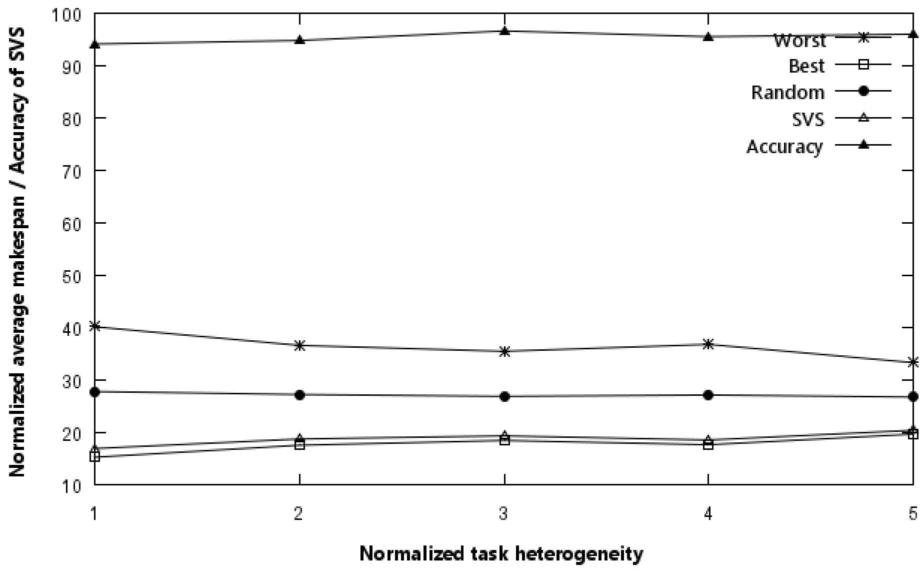

Figure 10.19: Normalized makespan and Accuracy by task heterogeneity

Table 10.1: Combinations of input and training distributions functions

Training distribution	Input distribution
Uniform	Uniform
Uniform	Normal
Normal	Normal
Normal	Uniform

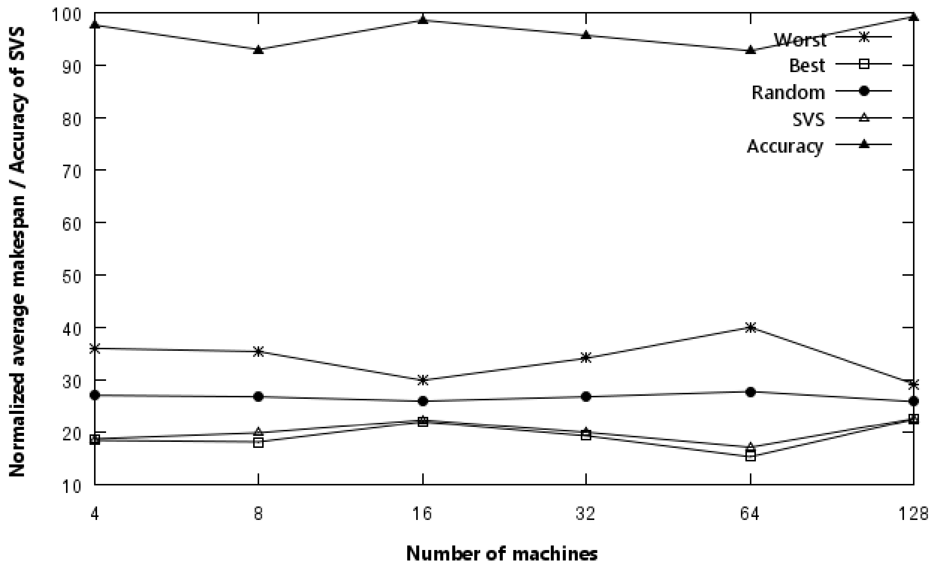

Figure 10.20: Normalized makespan and Accuracy by no. of machines

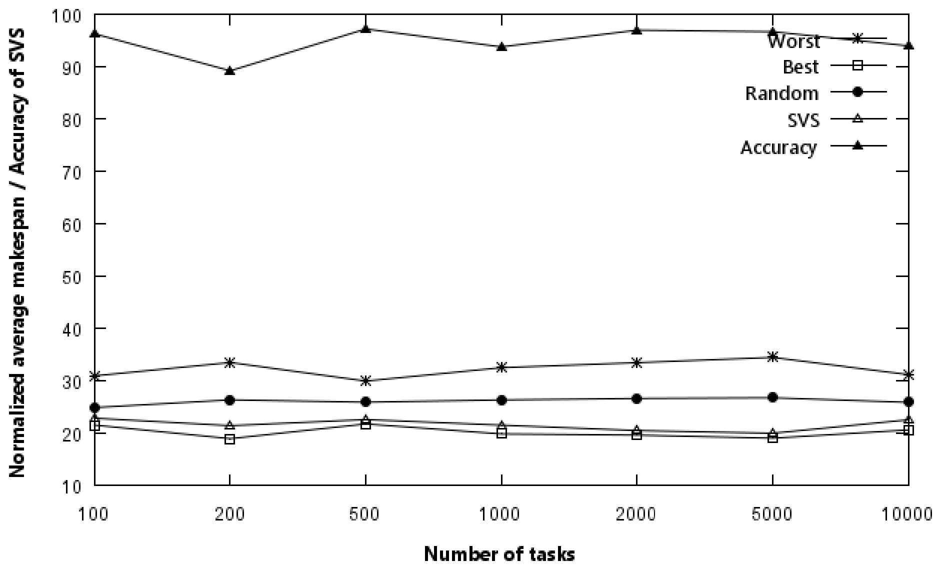

Figure 10.21: Normalized makespan and Accuracy by no. of tasks

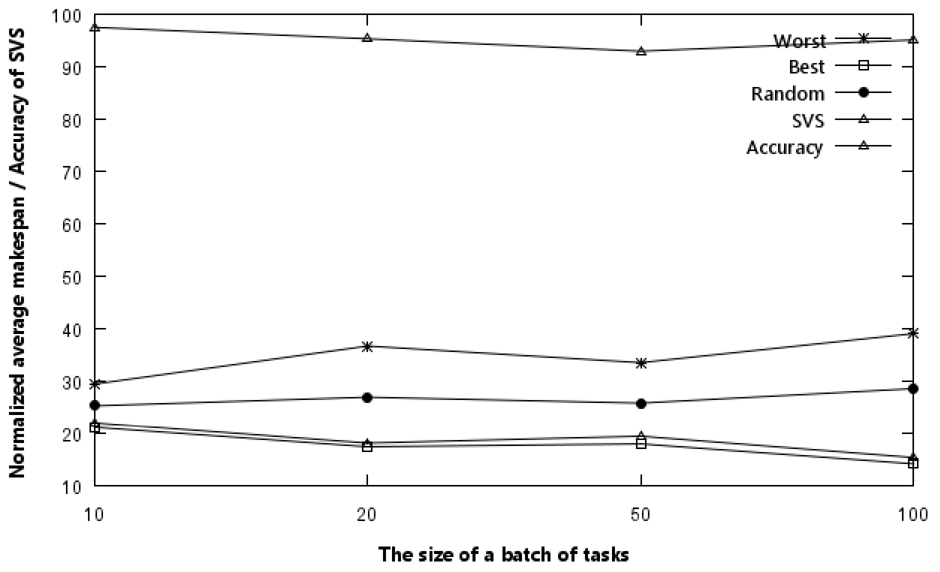

Figure 10.22: Normalized makespan and Accuracy by batch size

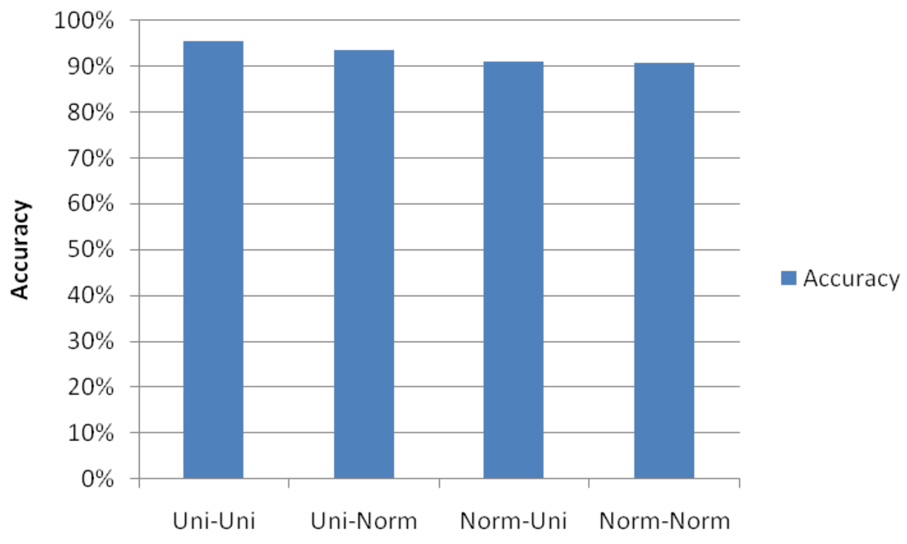

Figure 10.23: Accuracy by random number generator

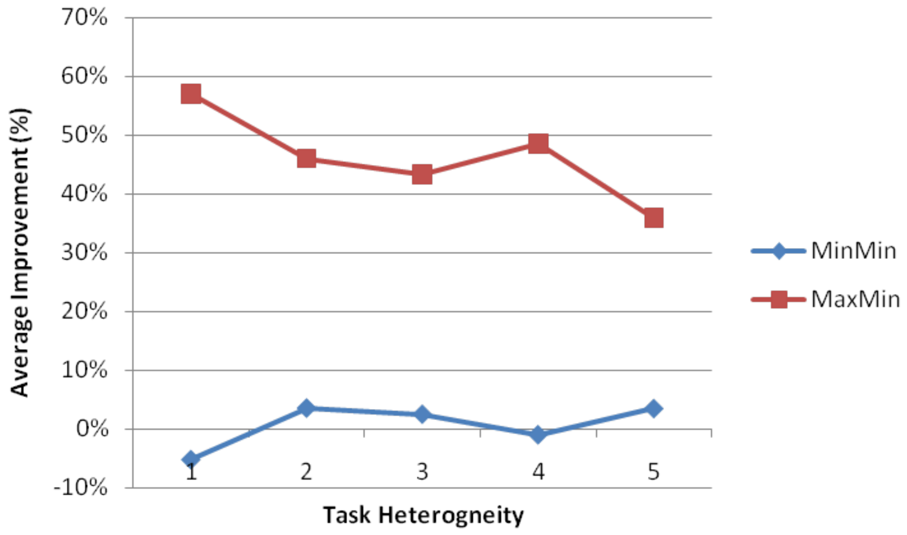

Figure 10.24: Average improvement over MinMin and MaxMin by Task Heterogeneity in Uni-Uni

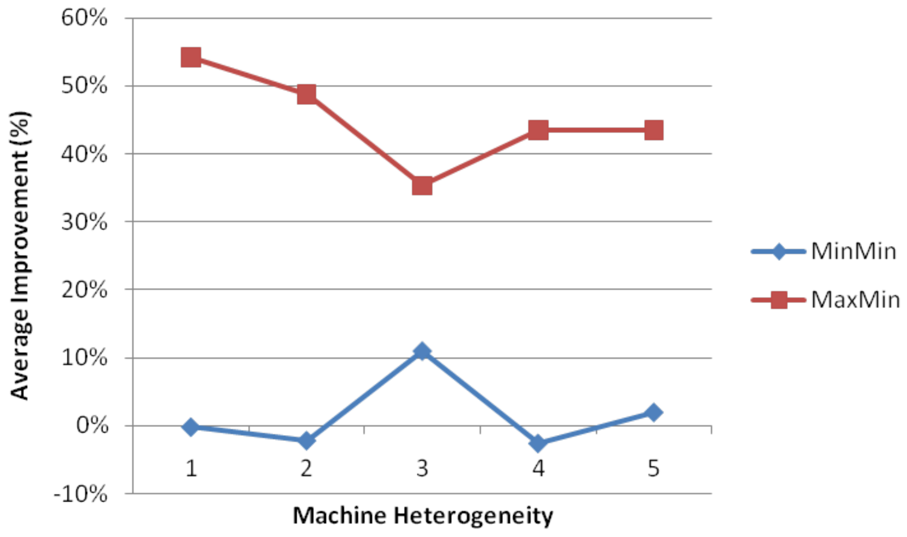

Figure 10.25: Average improvement over MinMin and MaxMin by Machine Heterogeneity in Uni-Uni

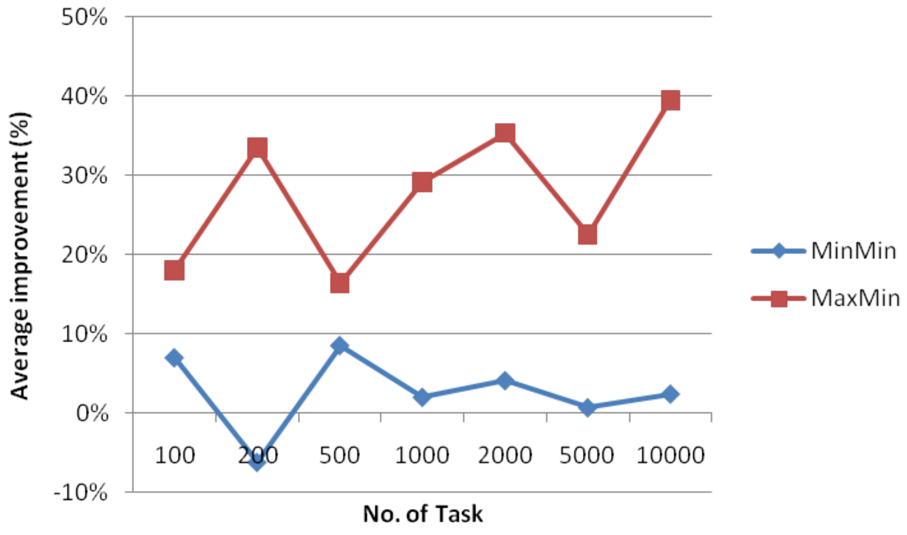

Figure 10.26: Average improvement over MinMin and MaxMin by No. of Task in Uni-Uni

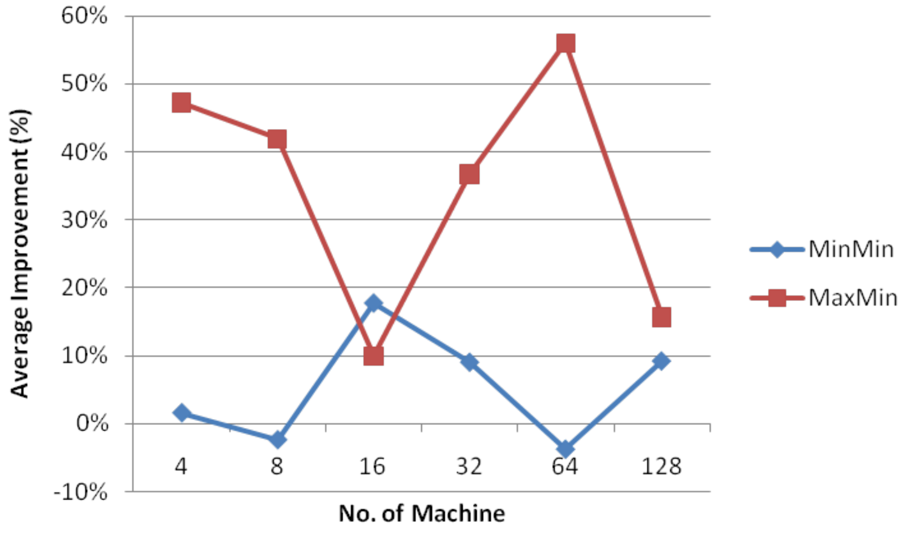

Figure 10.27: Average improvement over MinMin and MaxMin by No. of Machine in Uni-Uni

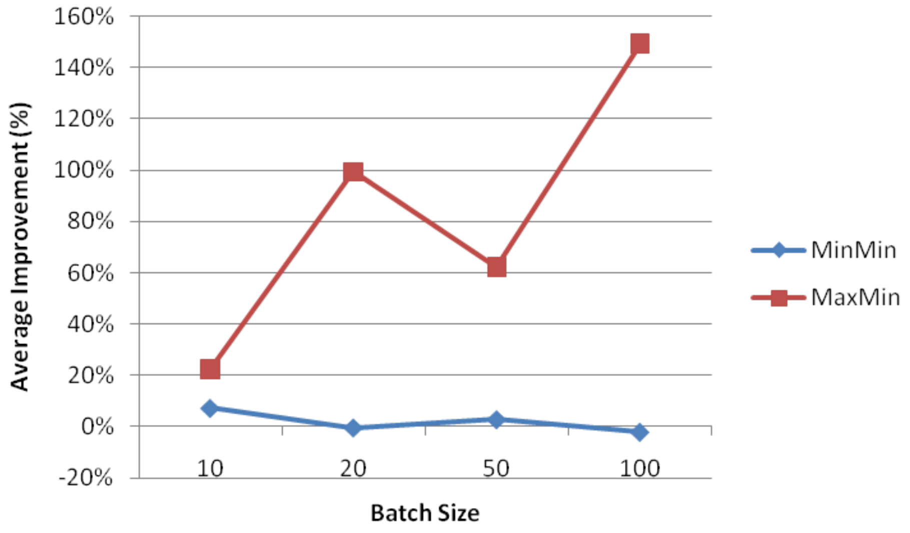

Figure 10.28: Average improvement over MinMin and MaxMin by Batch size in Uni-Uni

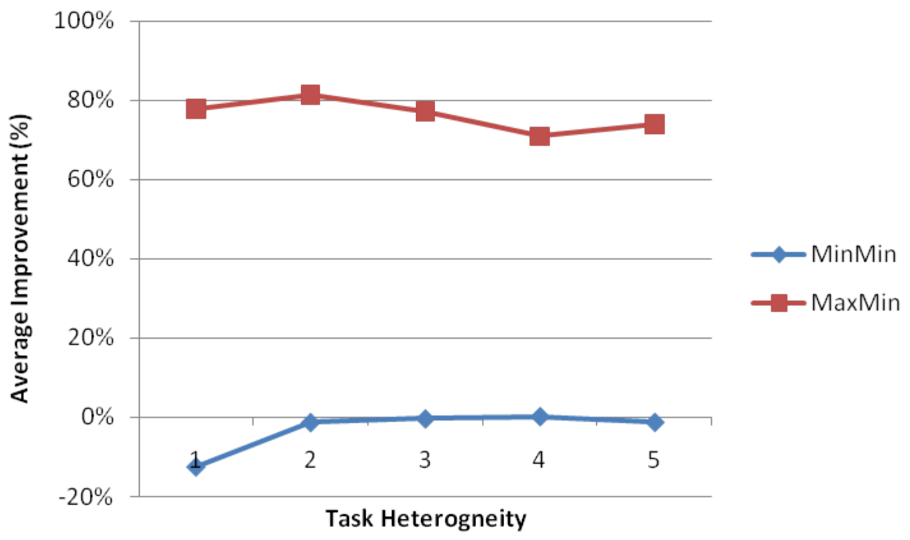

Figure 10.29: Average improvement over MinMin and MaxMin by Task Heterogeneity in Uni-Norm

Figs. 10.34 - 10.38 show the average improvement over *MinMin* and *MaxMin* for Norm-Uni combination. In this combination no improvements over *MinMin* heuristic are found since *MinMin* outperforms *MaxMin* for all criteria. Figs. 10.39 - 10.43 show the average improvement over *MinMin* and *MaxMin* for Norm-Norm combination. Fig. 10.40 shows the average improvement over *MinMin* and *MaxMin* by Machine heterogeneity where it is gained ideally by best using of the performance difference between *MinMin* and *MaxMin* heuristics.

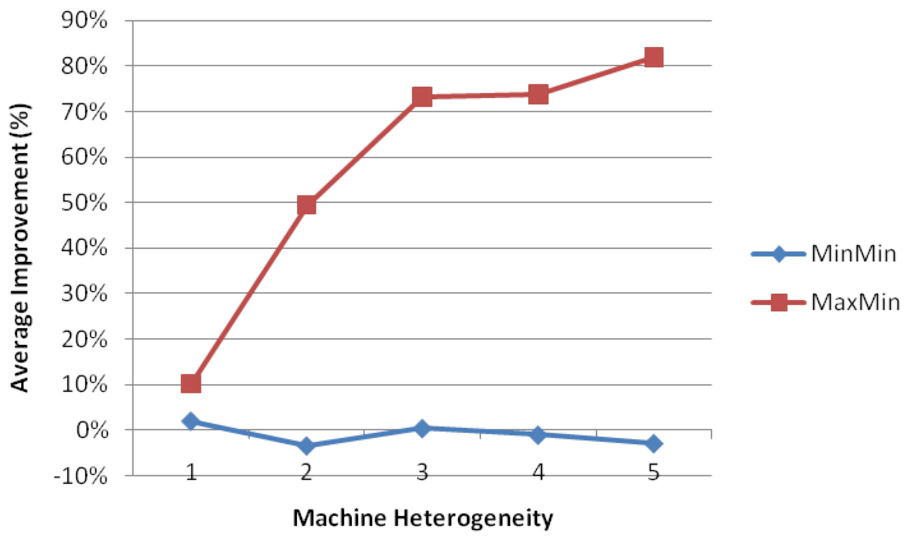

Figure 10.30: Average improvement over MinMin and MaxMin by Machine Heterogeneity in Uni-Norm

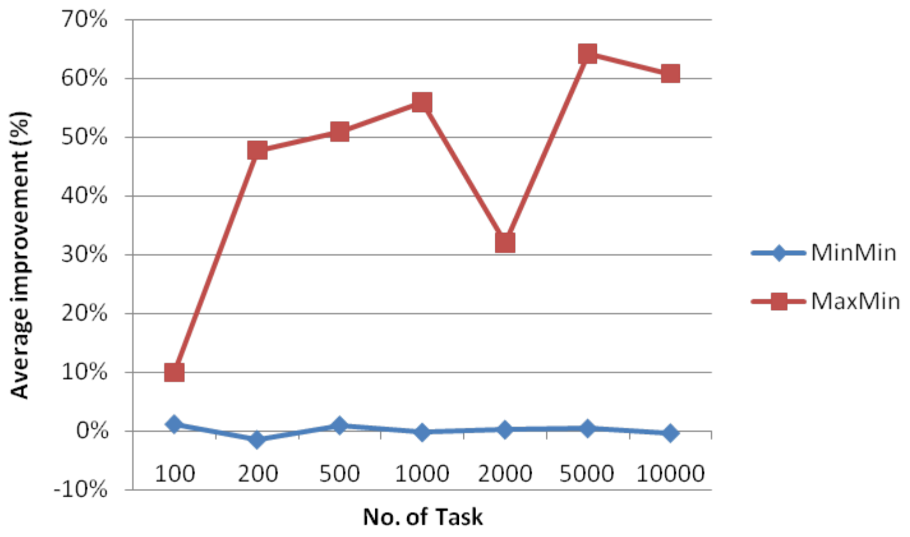

Figure 10.31: Average improvement over MinMin and MaxMin by No. of Task in Uni-Norm

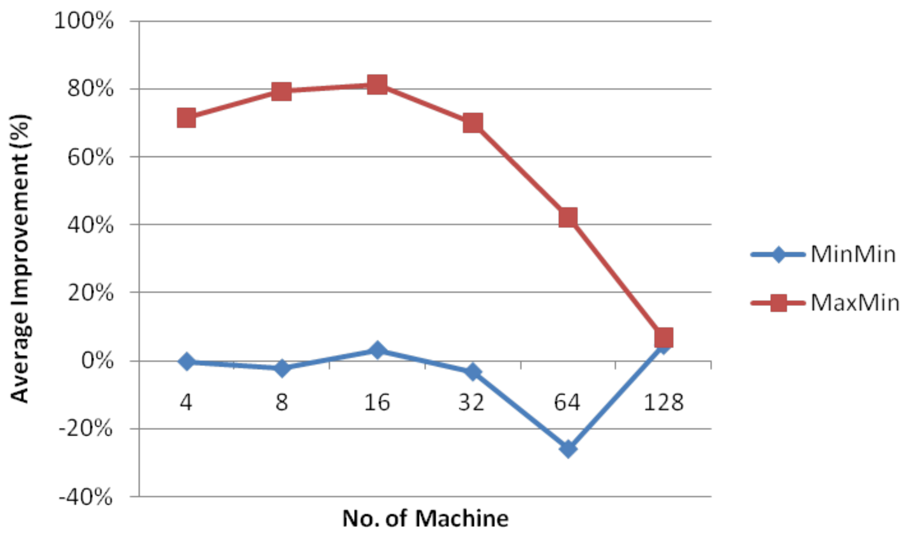

Figure 10.32: Average improvement over MinMin and MaxMin by No. of Machine in Uni-Norm

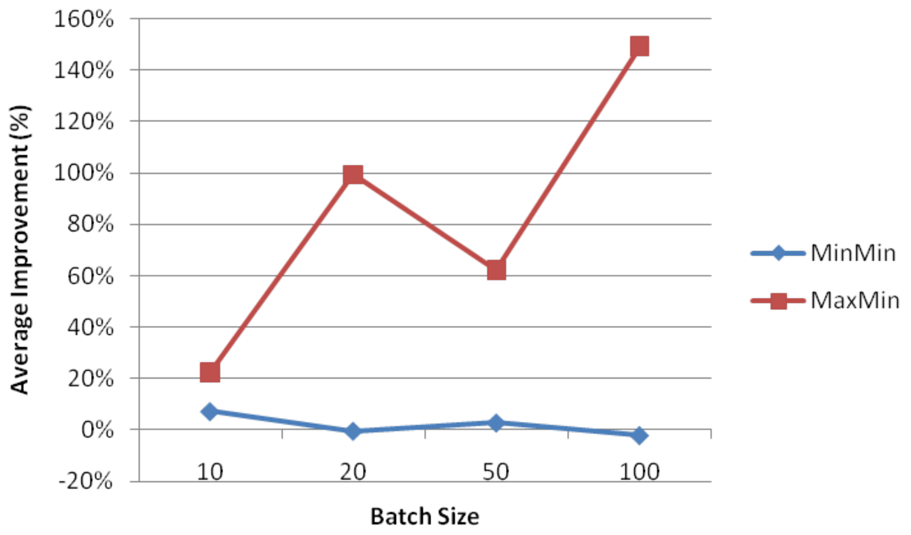

Figure 10.33: Average improvement over MinMin and MaxMin by Batch size in Uni-Norm

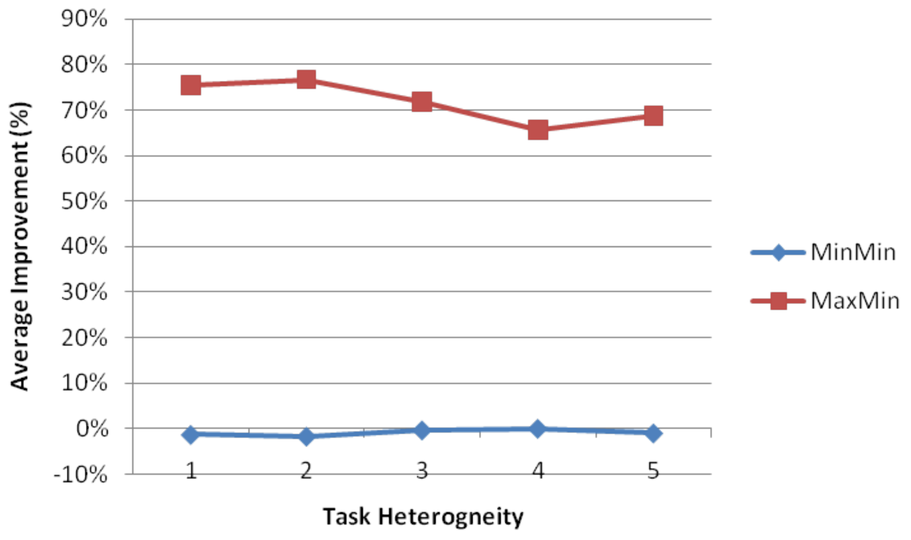

Figure 10.34: Average improvement over MinMin and MaxMin by Task Heterogeneity in Norm-Uni

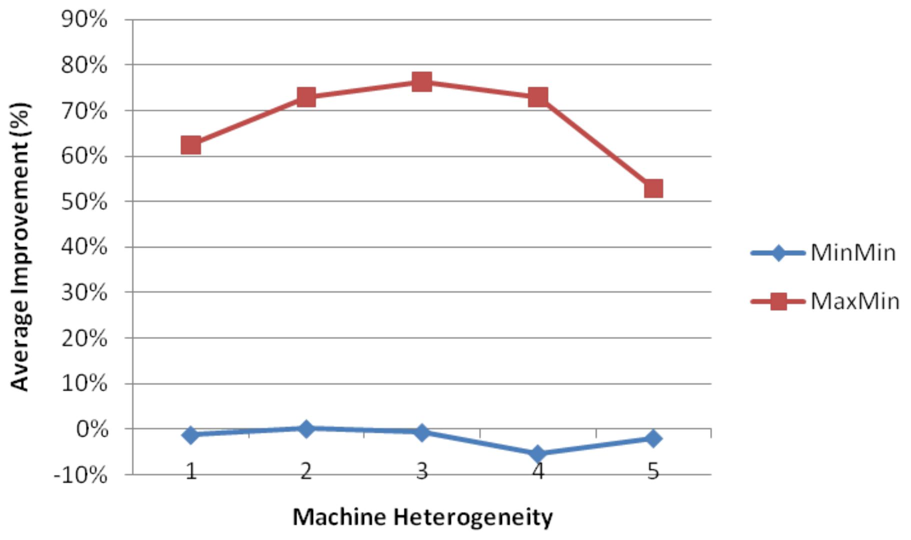

Figure 10.35: Average improvement over MinMin and MaxMin by Machine Heterogeneity in Norm-Uni

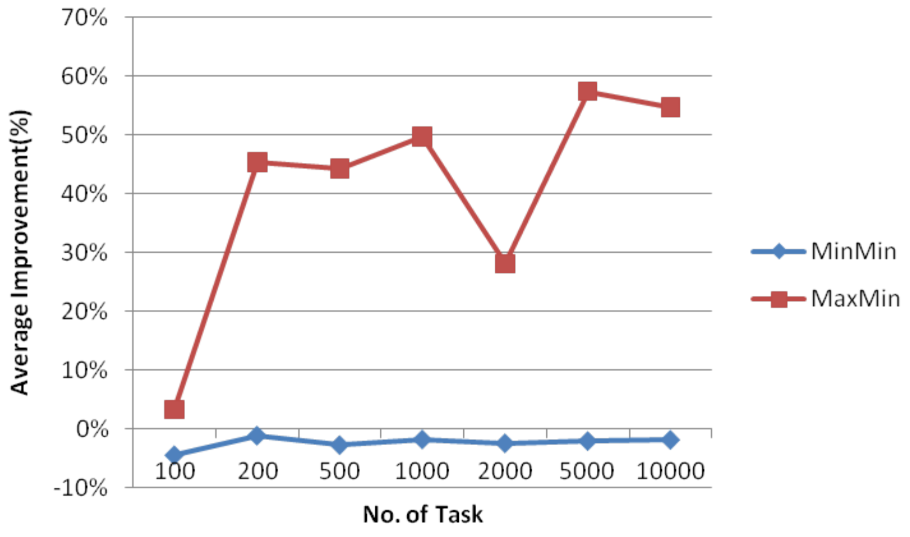

Figure 10.36: Average improvement over MinMin and MaxMin by No. of Task in Norm-Uni

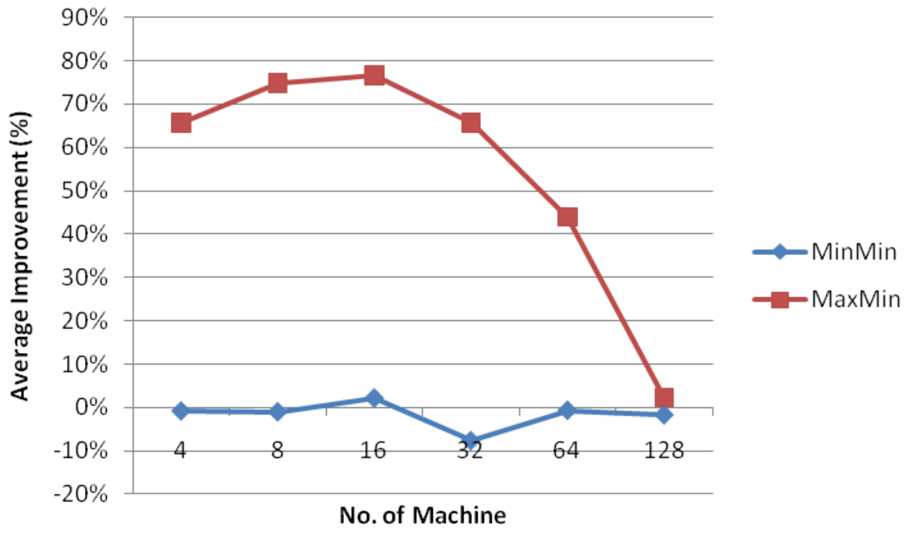

Figure 10.37: Average improvement over MinMin and MaxMin by No. of Machine in Norm-Uni

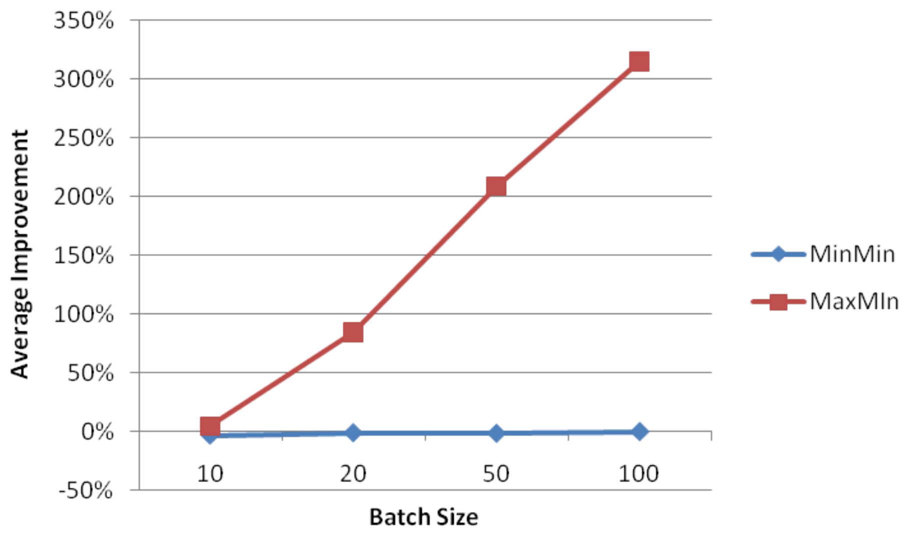

Figure 10.38: Average improvement over MinMin and MaxMin by Batch size in Norm-Uni

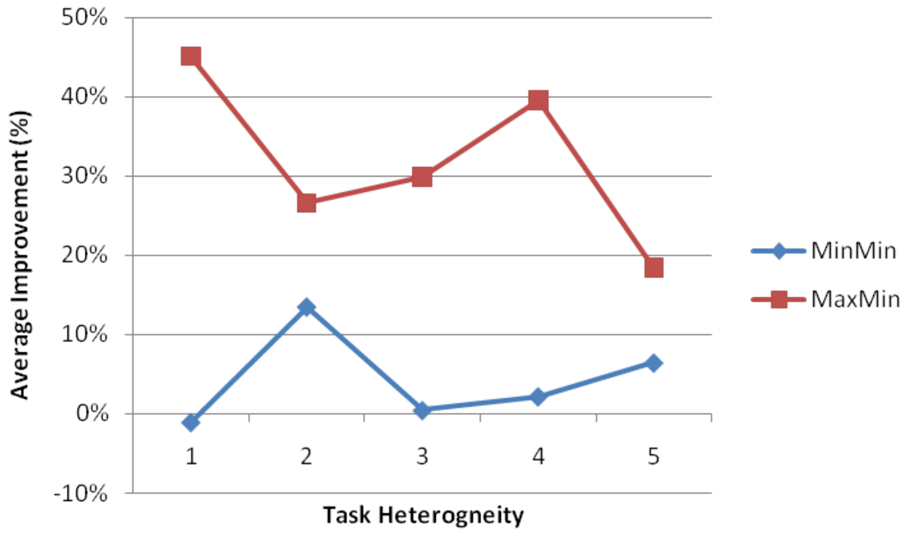

Figure 10.39: Average improvement over MinMin and MaxMin by Task Heterogeneity in Norm-Norm

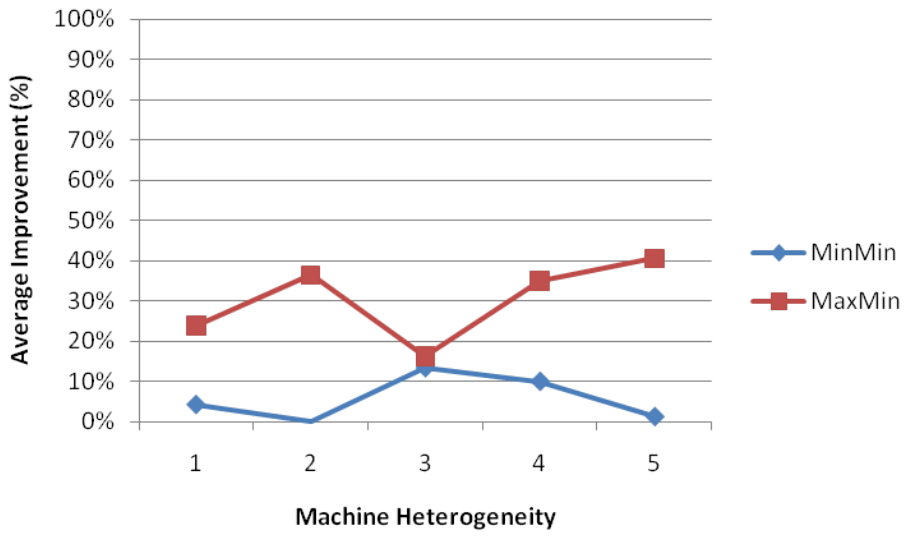

Figure 10.40: Average improvement over MinMin and MaxMin by Machine Heterogeneity in Norm-Norm

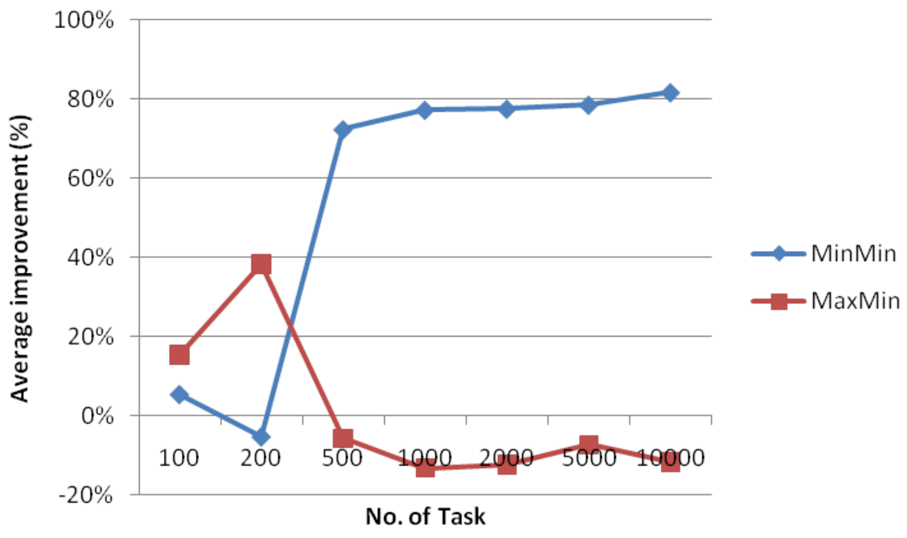

Figure 10.41: Average improvement over MinMin and MaxMin by No. of Task in Norm-Norm

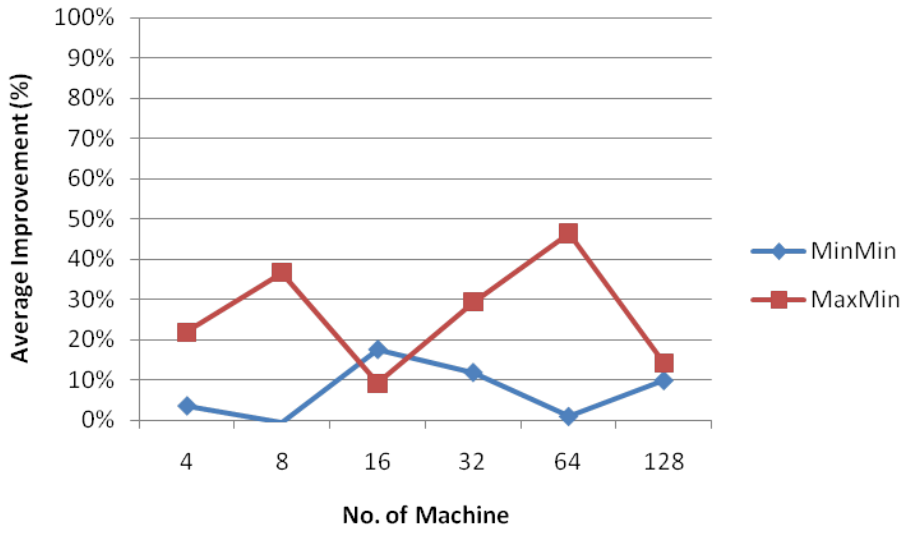

Figure 10.42: Average improvement over MinMin and MaxMin by No. of Machine in Norm-Norm

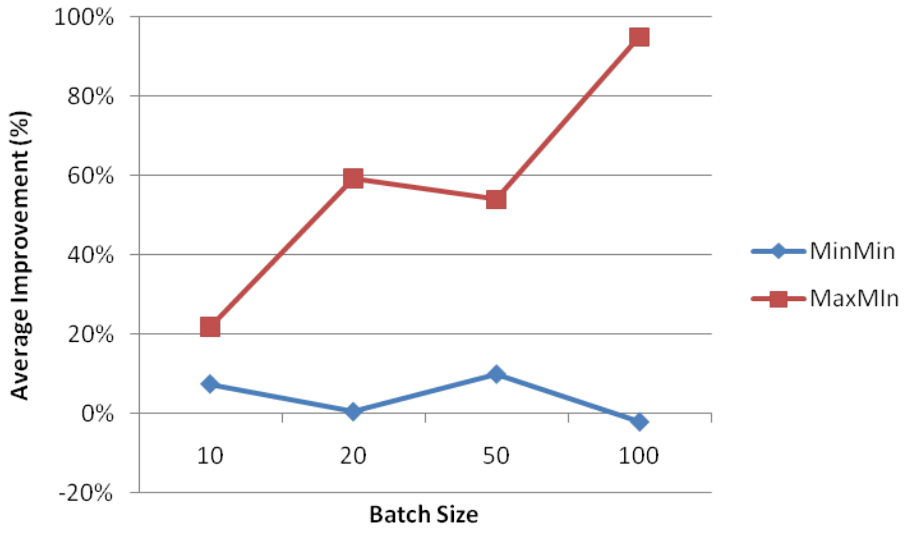

Figure 10.43: Average improvement over MinMin and MaxMin by Batch size in Norm-Norm

Chapter 11

Conclusion

Over the last four decades there have been many improvements in computing technology with the availability of cheaper, faster, and more reliable electronic components. These developments have enabled fast-processor computers to solve computation-demanding problems that would otherwise be impossible. However, since electronic processing speeds began to approach limitations imposed by the laws of physics, a new computing paradigm of parallel and distributed computing (PDC), allowed the transition from sequential to parallel processing, overcoming the limitations of a single processor. In this work, we addressed mechanisms concerned with orchestrating computing resources in PDC system to maximize system performance. We focused on describing the task mapping problem of mapping tasks onto heterogeneous systems and relevant architectures of PDC systems. Mapping independent tasks onto heterogeneous machines is known to be NP-complete. Therefore, most approaches were to create specific new heuristics, or to tailor existing heuristics to fit the problem. In this research, we applied machine learning techniques, particularly support vector machines (SVM), to solve the mapping problem. We formulated the mapping problem into a binary classification problem in which either one of two available heuristics is selected to map tasks onto machines. SVM is based on the principle of structural risk minimization, and has shown better performance than empirical risk minimization-based learning machines such as neural networks. We applied SVM to recognize certain patterns from a batch of tasks represented in the form of an expected execution time table, and to classify the batch of tasks into either class of the two heuristics. To do this, an SVM was implemented which is able to learn how to classify unknown input into one of the predefined classes. The process of implementing an SVM requires solving a convex optimization problem by finding an optimal hyperplane separating the sample data with maximal margin. In order to find the optimal hyperplane, we have to determine the value of parameters which correspond to each sample instance to result in maximal margin. Ultimately, the optimal hyperplane consisting of support vectors serves as a decision boundary for any unknown input. We implemented an SVM by solving the optimization problem using particle swarm optimization (PSO). It is known that the accuracy of a SVM depends on the ability of adjusting its capacity to the problem domain, thus resulting in its high accuracy. We achieved the best capacity of an SVM for the problem domain using a PSO. CPSO is a modification of the PSO, tailored to prevent particles from prematurely converging to local solutions by providing diversity so that the particles can escape from local optima.

This work presented a novel support vector scheduler (SVS) which uses a SVM to schedule tasks onto machines in a heterogeneous computing (HC) system. The SVS used SVM to selectively choose either one of two available heuristics, the *MinMin* and *MaxMin*. We have simulated the SVS consisting of the *MinMin* and *MaxMin* heuristics. The SVS was compared against the best, the worst, and random selection. The results of simulation show that the performance of SVS is very close to the best selection that can be achieved theoretically, achieving high improvement over the random selection.

From this study, we strongly suggest that our approach using machine learning techniques is plausible, and the SVS is very efficient for the following reasons. First, SVS is adaptive to the dynamic environment since it considers the performance difference between the *MinMin* and *MaxMin* by system variables. Next, the SVS is reliable since it has the inherent reliability from the use of conventional heuristics which have been already verified in many applications. Finally, the SVS is very effective in that it can create a new heuristic quickly by combining existing heuristics.

In the future, we will extend SVS to a variety of problem domains that require various objective functions and use more number of heuristics using a multi-class SVM.

Bibliography

[1] R. Nevatia, “Heterogeneous computing for vision,” Heterogeneous Computing Workshop, pp. 37-42, 26 April 1994.

[2] L. Wang, H. J. Siegel, V. P. Royehowdhury, and A. A. Maciejewski, “Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach,” Journal of parallel and distributed computing, vol. 47, pp. 8-22, 1997.

[3] M. Maheswaran, S. Ali, H. J. Siegel, D. A. Hensgen, and R. F. Freund, “Dynamic mapping of a class of independent tasks onto heterogeneous computing systems, ” Journal of parallel and distributed computing, vol. 59, pp. 107-131, 1999.

[4] T. Joachims, “Text Categorization with support vector machines: Learning with many relevant features,” Proceedings of the 10^{th} European Conference on Machine Learning, pp. 137-142, 1998.

[5] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. L. Wang, “Heterogeneous computing: challenges and opportunities,” Computer, vol. 26, pp. 18-27, June 1993.

[6] I. Ekmecic, I. Tartalja, and V. Milutinovic, “A survey of heterogeneous computing: concepts and systems,” Proceedings of the IEEE, vol. 84, pp. 1127-1144, 1996.

[7] I. Ekmeci, “EM3: A taxonomy of heterogeneous computing systems,” Computer, pp. 6870, 1995.

[8] M. Maheswaran and H. J. Siegel, “A dynamic matching and scheduling algorithm for heterogeneous computing systems,” Proceedings of the 7^{th} Heterogeneous Computing Workshop, pp. 57, 1998.

[9] H. D. Karatza and R. C. Hilzer, “Load sharing in heterogeneous distributed systems,” Winter Simulation Conference, vol. 1, pp. 489-496, 2002.

[10] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, and J. D. Lima, “Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet,” Heterogeneous Computing Workshop, pp. 184-199, 1998.

[11] T. D. Braun, H. J. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, and M. D. Theys, “A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems,” Proceedings of 17^{th} IEEE Symposium on Reliable Distributed Systems, pp. 330-335, 1998.

[12] A. J. Page and T. J. Naughton, "Dynamic task scheduling using genetic algorithms for heterogeneous distributed Computing," Proceedings of 19^{th} IEEE International in Parallel and Distributed Processing Symposium, pp. 189a, 2005.

[13] Yi-Hung Liu, Han-Pang Huang, and Yu-Sheng Lin, "Dynamic scheduling of flexible manufacturing system using support vector machines," IEEE International Conference on Automation Science and Engineering, pp. 387-392, 1-2 August 2005.

[14] V. Vapnik, "The nature of statistical learning theory," Springer-verlag, New York, 1995.

[15] C. Cortes and V. Vapnik, "Support vector networks," Machine Learning, vol. 20 pages 273-297, 1995.

[16] V.Blanz, B. Schölkopf, H. H. Bülthoff, C. Burges, V. Vapnik, and T. Vetter, "Comparison of view-based object recognition algorithms using realistic 3D models," In Proceedings of the 1996 international Conference on Artificial Neural Networks (July 16 - 19, 1996). C. v. Malsburg, W. v. Seelen, J. C. Vorbrüggen, and B. Sendhoff, Eds. Lecture Notes In Computer Science, vol. 1112, Springer-Verlag, London, 251-256.

[17] M. Schmidt, "Identifying speaker with support vector networks," Proceedings of Interface '96, Sydney, 1996.

[18] E. Osuna, R. Freund, and F. Girosi, "Training support vector machines: an application to face detection," In IEEE Conference on Computer Vision and Pattern Recognition, pages 130-136, 1997.

[19] C. J. C. Burges, "A tutorial on support vector machines for pattern recognition," Data mining and knowledge discovery, vol. 2, pp. 121-167, 1998.

[20] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, and D. Hensgen, "A Comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," Journal of Parallel and Distributed Computing, vol. 61, pp. 810-837, 2001.

[21] J. Choi, C. Y. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose, "Thermal-aware task scheduling at the system software level," Proceedings of the International Symposium on Low Power Electronics and Design, 2007.

[22] A. K. Coskun, T. S. Rosing, and K. Whisnant, "Temperature aware task scheduling in MPSoCs," Proceedings of the conference on Design, automation and test in Europe, pp. 1659-1664, 2007.

[23] A. Merkel and F. Bellosa, "Task activity vectors: a new metric for temperature-aware scheduling," Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems, pp. 1-12, 2008.

[24] R. Freund and H. J. Siegel, "Heterogeneous processing," IEEE Computer, vol. 26, no. 6, pp. 13-17, June 1993.

[25] Mary M. Eshaghian, "Heterogeneous computing," Artech House, Norwood, MA, 1996.

[26] H.J. Siegel, J.K. Antonio, R.C. Metzger, M. Tan, and Y.A. Li, "Heterogeneous computing," Parallel and Distributed Computing Handbook, A.Y. Zomaya, ed., pp. 725-761. New York: McGraw-Hill, 1996.

[27] H. J. Siegel, H. G. Dietz, and J. K. Antonio, "Software support for heterogeneous computing," The Computer Science and Engineering Handbook, A. B. Tucker, Jr., ed., CRC Press, Boca Raton, pp. 1886-1909, FL, 1997

[28] D. Fernandez-Baca, "Allocating modules to processors in a distributed system," IEEE Transactions on Software Engineering, vol. se-15, no. 11, pp. 1427-1436, November 1989.

[29] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on non-identical processors," Journal of the ACM, vol. 24, no. 2, pp. 280-289, April 1977.

[30] A. M. Mehta, J. Smith, H. J. Siegel, A. A. Maciejewski, A. Jayaseelan, and B. Ye, "Dynamic resource management heuristics for minimizing makespan while maintaining an acceptable level of robustness in an uncertain environment," Proceedings of the 12^{th} International Conference on Parallel and Distributed Systems, vol. 1, pp. 107-114, 2006.

[31] J. Kennedy and R. Eberhart, "Particle swarm optimization," Proceedings of IEEE International Conference on Neural Networks, vol. 4, pp. 1942-1948, November/December 1995.

[32] A. Bricker, M. Litzkow, and M. Livny, "Condor technical summary. University of Wisconsin, Madison," version 4.1b edition, January 1992.

[33] R. A. Henry, N. S. Flann, and D. W. Watson, "A massively parallel SIMD algorithm for combinatorial optimization," In The 1996 International Conference on Parallel Processing, vol. 2, pages 46-49, August 1996.

[34] J. A. Kaplan and M. L. Nelson, "A comparison of queuing, cluster and distributed computing systems," Technical Report NASA TM 109025, NASA Langley Research Center, June 1994.

[35] M. A. Iverson and F. Ozguner, "Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment," Proceedings of 7^{th} Heterogeneous Computing Workshop (HCW 98), pp. 70-78, 30 March 1998

[36] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," IEEE Transactions on Software Engineering, vol. 14, pp. 141-154, 1988.

[37] R. Mirchandaney, D. Towsley, and J. A. Stankovic, "Adaptive load sharing in heterogeneous systems," 1989.

[38] E. K. Burke, S. Petrovic, and R. Qu, "Case-based heuristic selection for timetabling problems," Journal of Scheduling, vol. 9, pp. 115-132, 2006.

[39] C. P. Ravikumar and V. Naresh, “Heuristic and neural algorithms for mapping tasks to a reconfigurable array,” Microprocessing and Microprogramming, vol. 41, pp. 137-151, 1995.

[40] A. Arbelaez, Y. Hamadi, and M. Sebag, “Online heuristic selection in constraint programming,” HAL - CCSD, 2009.

[41] W. T. Reeves, “Particle Systems-a Technique for Modeling a Class of Fuzzy Objects,” ACM Trans Graph, vol. 2, pages 91-108, 1983.

[42] Craig W. Reynolds, “Flocks, herds and schools: A distributed behavioral model,” SIGGRAPH Comput. Graph, vol. 21, pages 25-34, 1987.

[43] J. Riget and J. S. Vesterstrom, “A Diversity-Guided Particle Swarm Optimizer - the ARPSO,” 2002.

[44] M. J. Flynn, “Parallel Architectures,” ACM Computing Surveys, vol. 28, no. 1, pp. 19-25, March 1996.

[45] T. Kohonen, “The self-organizing map,” Neurocomputing, vol. 21, no. 2, pages 1-6, 6 November 1993.

[46] Thomas T. Kwan, Robert E. McGrath, and Daniel A. Reed, “EM3: A Taxonomy of Heterogeneous Computing Systems,” Computer, vol. 28, no. 12, pages 68-70, 1995.

[47] V. M. Lo, “Heuristic Algorithms for Task Assignment in Distributed Systems,” IEEE Trans. on Computers, vol. 37, no. 11, pages 1384- 1397, November 1988.

[48] N. S. Bowen, C. N. Nikolaou, and A. Ghafoor, “On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems,” IEEE lkans. on Computers, vol. 41, no. 3, pp. 257-273, March 1992.

[49] W. W. Chu, L. J. Holloway, M-T. Lan, and K. Efe, “Task Allocation in Distributed Data Processing,” IEEE Computer, pages 57-69, November 1980.

[50] S. S. Yau and V. R. Satish, “A Task Allocation Algorithm for Distributed Computing Systems,” COMPSAC, pages 336, 1993.

[51] C-C. Shen and W-H. Tsai, “A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion,” IEEE Trans. on Computers, vol. C-34, no. 3, pages 197-203, March 1985.

[52] H. S. Stone, “Multiprocessor scheduling with the aid of network flow algorithms,” IEEE Trans. Software Eng., vol. SE-3, pp. 85-93, January 1977.

[53] Syam Menon, “Effective Reformulations for Task Allocation in Distributed Systems with a Large Number of Communicating Tasks,” IEEE Trans. on Knowl. and Data Eng, vol. 16, no. 12, pages 1497-1508, 2004.

[54] S. S. Yau and I. Wiharja, “An Approach to Module Distribution for the Design of Embedded Distributed Software Systems”, Information Sciences, vol. 56, pages 1-22, 1991.

[55] S. S. Yau, D.-H. Bae, and G. Pour, “A Partitioning Approach for Object-Oriented Software Development for Parallel Processing Systems,” Proceedings of 16^{th} Ann. Int’l Computer Software and Application Conf. (COMPSACSZ), pages 251-256, September 1992

[56] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky, “SETI@home-massively distributed computing for SETI,” Computing in Science & Engineering In Computing in Science & Engineering, vol. 3, no. 1, pp. 78-83, 2001.

[57] Shih-Nung Chen1, Jeffrey J.P. Tsai2, Chih-Wei Huang1, Rong-Ming Chen4, and Raymond C.K. Lin, “Using Distributed Computing Platform to Solve High Computing and Huge Data Processing Problems in Bioinformatics,” Proceedings of the 4^{th} IEEE Symposium on Bioinformatics and Bioengineering, 2004.

[58] J. Shurkin, “Engines of the Mind: A History of the Computer,” New York: W. W. Norton & Company, 1984.

[59] M. V. Wilkes, “Memoirs of a Computer Pioneer,” Cambridge, Mass.:MIT Press, 1985.

[60] William Aspray, “John von Neumann and the Origins of Modern Computing,” Cambridge: MIT Press, 1990.

[61] NSF, “Grand Challenge: High-Performance Computing and Communications,” Report, Committee on Physical, Mathematical, and Engineering Sciences, Washington, D.C.: U.S. Office of Science and Technology Policy, National Science Foundation, 1992.

[62] M.D. Godfrey and D.F. Hendry, “The computer as von Neumann planned it,” IEEE Annals of the History of Computing, vol. 15, no. 1, pp. 11-21, 1993.

[63] H.J. Siegel, T. Schwedreski, W. G. Nation, J. B. Armstrong, L. Wang, J. T. Kuehn, R. Gupta, M. D. Allemang, D. G. Meyer, and D. G. Watson, “Chapter 3, The design and prototyping of the PASM reconfigurable parallel processing system,” In Parallel Computing: Paradigms and Applications, ed. A. Y. Zomaya London: International Thomson Computer Press, pp. 78-114.

[64] D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface,” San Mateo, Calif.: Morgan Kaufmann Publishers, 1994.

[65] H. S. Stone, “High-Performance Computer Architectures,” 2d e. Reading, Mass.: Addison-Wesley, 1990.

[66] D. I. Moldovan, “Parallel Processing: From Applications to Systems,” San Mateo, Calif.: Morgan Kaufmann Publishers, 1993.

[67] K. Hwang, “Advanced Computer Architecture: Parallelism, Scalability, and Programmability,” New York: McGraw-Hill, 1993.

[68] H. T. Kung, “Why systolic architectures?,” IEEE Computer, pp. 37-46, 1982.

[69] B. Wilkinson and M. Allen, “Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers,” Prentice Hall, 1998.

[70] C.V. Ramamoorthy and H. F. Li, “Pipeline architecture,” ACM Computing Surveys, vol. 9, no. 1, pp. 61-102, 1977.

[71] Arthur. H. Veen, “Dataflow machine architecture,” ACM Computing Surveys, vol. 18, no. 4, 1986.

[72] G. Langholz, J. Francioni, and A. Kandel, “Elements of Computer Organization,” Englewood Cliffs, N.J.: Prentice-Hall, 1989.

[73] H.J. Siegel, “Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,” ed. 2. New York: McGraw-Hill, 1990.

[74] E. G. Coffman, “Computer and Job-Shop Scheduling Theory,” New York: John Wiley & Sons, 1976.

[75] M. R. Garey and D. S. Johnson, “Computers and Intractability: A Guide to the Theory of NP Completeness,” San Francisco: W. H. Freeman, 1979.

[76] T. G. Lewis and H. El-Rewini, “Introduction to Parallel Computing,” Englewood Cliffs, N.J.: Prentice Hall, 1992.

[77] J. Ullman, “NP-complete scheduling problems,” Journal of Computer and System Sciences, vol. 10, pp. 384-393, 1975.

[78] E. Dekel, E. Nassimi, and S. Sahni, “Parallel matrix and graph algorithms,” SIAM Journal of Computing, vol. 1, 657-675, 1981.

[79] W. M. Gentleman, “Some complexity results for matrix computations on parallel processors,” Journal of the ACM, vol. 25, pages 112-115, 1978.

[80] H. J. Siegel, B. Armstrong, and D. W. Watson, “Mapping computer-vision-related tasks onto reconfigurable parallel-processing systems,” IEEE Computer, vol. 25, no. 2, pages 54-63.

[81] H. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley Jr., and S. D. Smith, “PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition,” IEEE Transactions on Computers, vol. c-30, no. 12, pages 934-947, 1981.

[82] K. E. Batcher, “Design of a massively parallel processor,” IEEE Transactions on Computers, vol. c-29, no. 9, pages 836-844, 1980.

[83] R. F. Freund, “SuperC or distributed heterogeneous HPC,” Computing Systems in Engineering, vol. 2, no. 4, pages 349-355, 1991.

[84] S. A. Fineberg, T. L. Casavant, and H. J. Siegel, “Experimental analysis of a mixed-mode parallel architecture using bitonic sequence sorting,” J. Parallel and Distributed Computing, vol. 11, no. 3, pages 239-251, 1991.

[85] D. W. Watson, J. K. Antonio, H. J. Siegel, and M. J. Atallah, “Static program decomposition among machines in an SIMD/SPRM heterogeneous environment with nonconstant mode switching costs,” Proceedings of the Heterogeneous Computing Workshop, pp. 58-65, 1994.

[86] M. J. Flynn, “Very high-speed computing systems,” Proceedings of the IEEE, vol. 54, no. 12, pp. 1901-1902, 1966.

[87] R. F. Freund, “Optimal selection theory for superconcurrency,” Proceedings of Supercomputing ’89, pp. 699-703, 1989.

[88] T. L. Casavant and J. G. Kuhl, “A taxonomy of scheduling in general-purpose distributed computing systems,” IEEE Trans. Software Engineering, vol. 14, no. 2, pp. 141-154, 1988.

[89] H.E. Bal, J.G Steiner, and A. S. Tanenbaum, “Programming languages for distributed computing systems,” ACM Comput. Surv, vol. 21, no. 3, 1989.

[90] U. Brinschulte, A. von Renteln, and M. Pacher, “Measuring the quality of an artificial hormone system based task mapping,” Autonomics ’08: Proceedings of the 2nd International Conference on Autonomic Computing and Communication Systems, pp. 1-8, 2008.

[91] J. Beck and D. Siewiorek, “Modeling Multicomputer Task Allocation as a Vector Packing Problem,” ISSS ’96: Proceedings of the 9^{th} international symposium on System synthesis, pp. 115, 1996.

[92] A. Billionnet, M. C. Costa, and A. Sutter, “An efficient algorithm for a task allocation problem,” ACM, vol. 39, no. 3, pages 502-518, 1992.

[93] B. Hong and V. K. Prasanna, ”Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput,“Proceedings of 18^{th} International conference on Parallel and Distributed Processing Symposium, pp. 52, 2004.

[94] A.S. Khalifa, R.A. Ammar, T.A. Fegrany, and M.E. Khalifa, “A Preemptive Version of the Min-Min Heuristic for Dynamically Mapping Meta-Tasks on a Distributed Heterogeneous Environment,” 2007 IEEE International Symposium on Signal Processing and Information Technology, pages 537-542, 2007.

[95] Ning Weng and Tilman Wolf, “Profiling and mapping of parallel workloads on network processors,” SAC ’05: Proceedings of the 2005 ACM symposium on Applied computing, pp. 890-896, 2005.

[96] A. Jose, “An approach to mapping parallel programs on hypercube multiprocessors,” Proceedings of the 7th Euromicro Workshop on Parallel and Distributed Processing (PDP ’99), pp. 221-225, 1999.

[97] Tingwei Chen, Bin Zhang, Xianwen Hao, and Yu Dai, “Task Scheduling in Grid Based on Particle Swarm Optimization,” The 5th International Symposium on Parallel and Distributed Computing (ISPDC ’06), pp. 238-245, 6-9 July 2006.

[98] Yuanqiang Huang, Depei Qian, Zhongzhi Luan, Yongjian Wang, Zhongxin Wu, and Bingheng Yan, “EOMT: A Master-Slave Task Scheduling Strategy for Grid Environment,” 10th IEEE International Conference on High Performance Computing and Communications (HPCC ’08), pp. 226-233, 25-27 Sept. 2008.

[99] Leonard Kleinrock, “Nomadic computing—an opportunity,” SIGCOMM Comput. Commun. Rev., vol. 25, no. 1, pages 36-40, 1995.

[100] H. L. Gantt, “Work, Wages and Profit,” The Engineering Magazine, New York; republished as Work, wages, and profits, Hive Publishing Company, Easton, Pennsylvania, 1974.

[101] T. L. Sterling, “How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters Scientific and Engineering Computation,” Cambridge, Mass MIT Press, 1999.

[102] T. Joachims, “Text Categorization with Support Vector Machines: Learning with Many Relevant Features,” Proceedings of the European Conference on Machine Learning, 1998.

[103] B. Al-kazemi and C. K. Mohan, “Training feedforward neural networks using multiphase particle swarm optimization,” Proceedings of the 9th International Conference on Neural Information Processing, vol. 5, pp. 2615-2619, November 2002.

[104] O. Ibarra and C. E. Kim, “Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors,” J. ACM, vol. 24, no. 2, pages 280-289, NY, 1977.

[105] T. Krink, J.S. Vesterstromand, and J. Riget, “Particle swarm optimization with spatial particle extension,” Proceedings of the 2002 Congress on Evolutionary Computation, vol. 2, pp. 1474-1479, 2002.

[106] K. Veeramachaneni, T. Peram, C.K. Mohan, and L. A. Osadciw, “Optimization using particle swarms with near neighbor interactions,” Lecture Notes in Computer Science (LNCS) No. 2723: Proceedings of the Genetic and Evolutionary Computation Conference 2003 (GECCO 2003), pages 110-121, 2003.

[107] S. Baskar and P.N. Suganthan, “A novel concurrent particle swarm optimization,” Proceedings of the Congress on Evolutionary Computation, vol. 1, pages 792-796, June 2004.

[108] J. S. Taylor and N. Cristianini, “Kernel methods for pattern analysis, ” Cambridge University Press, Cambridge, UK, 2004.

[109] S. Y. Cho and K. Ho. Park, “Dynamic task assignment in heterogeneous linear array networks for metacomputing,” Proceedings of Heterogeneous Computing Workshop, pp. 66-71, April 1994.

[110] National Academy of Science-national Research Council Washington DC, ”Computing and Communications in the Extreme: Research for Crisis Management and Other Applications,” National Academies press, 1996.

[111] J. Wei, L. Guangbin, and L. Dong, “Elite Particle Swarm Optimization with mutation,” 7^{th} International Conference on System Simulation and Scientific Computing (ICSC 2008), pages 800-803, 2008.

[112] F. Wang and Y. Qiu, “A modified particle swarm optimizer with roulette selection operator,” Proceedings of 2005 IEEE International Conference on Natural Language Processing and Knowledge Engineering (IEEE NLP-KE ’05), pp. 765-768, 2005.

[113] Y. Shi and R. Eberhart, “A modified particle swarm optimizer,” Proceedings of 1998 IEEE International Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence, pp. 69-73, May 1998.

[114] Y. Park, S. Baskiyar, and K. Casey, “A Novel Adaptive Support Vector Machine based Task Scheduling,” Proceedings the 9^{th} International Conference on Parallel and Distributed Computing and Networks, Austria, 16-18 February 2010.

[115] J. Kennedy, “Evolutionary Computation,” Proceedings of the 2000 Congress, vol. 2, pp. 1507-1512, 2000.

[116] J. Holland, “Adaptation in natural and artificial systems,” MIT Press, 1998.

[117] Jang-Ho Seo, Chang-Hwan Im, Chang-Geun Heo, Jae-Kwang Kim, Hyun-Kyo Jung, and Cheol-Gyun Lee, “Multimodal function optimization based on particle swarm optimization,” IEEE Transactions on Magnetics, vol. 42, pages 1095-1098, April 2006.