Agent, Genetic Algorithm with Task Duplication based Scheduling Technique For Heterogenous Systems

by

Navdeep Singh Sidhu

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
August 2018

Thesis advisor Author

Dr. Parimala Thulasiraman **Navdeep Singh Sidhu**

Agent, Genetic Algorithm with Task Duplication based Scheduling Technique For Heterogenous Systems

Abstract

High-Performance Computing (HPC) is used to solve complex problems in parallel for increased performance. Over the past few years, parallelization has become more challenging with the many core general purpose systems and accelerators. One of the challenges is in better utilization of the resources available on these architectures through better task scheduling strategies. In this thesis I consider a distributed, heterogeneous network with general processing CPU based systems of varying speed and architectures. I propose an efficient mapping and scheduling of tasks to processors using agents to explore the network and Genetic Algorithm with Task Duplication Scheduling(GATDS) to schedule the tasks. The SIPS (Serial algorithms In Parallel System) framework is used to exploit parallelism using abstract syntax trees generated directly from the source code. This framework helps in automating the process of converting serial code for use in parallel systems, thus reducing the overhead of writing parallel code.

GATDS is compared with various scheduling strategies for task independent and task dependent problems. The performance of GATDS is comparable to the use of existing genetic algorithms for task independent problems. For inter-dependant tasks, the proposed technique matches or performs better than the traditional Chunk scheduler and genetic algorithm 75 % of the time. GATDS also provides better resource utilization.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

I would like to begin by thanking my advisor Dr. Parimala Thulasiraman for mentoring me and helping me in my thesis. Her expert guidance, consistent motivation and constructive criticism helped me overcome numerous challenges that i faced during my research.

I am very grateful to my committee members, Dr. Jun Cai and Dr. Peter Graham for their valuable comments, time and guidance.

I am thankful to my family, for their priceless love and support.

I am also thankful to my mentor Dr. Anil Kumar who introduced me to High Performance Computing (HPC).

Special thanks to my mentor, friend and big brother Raminder Pal Gill for his continuous motivation which always inspired me to push limits and achieve higher goals.

Thanks to all my friends Guramrit Singh, Anmol Sekhon, Arshdeep Samra, Karanjeet Singh Rai, Anurag and Muskaan for not disturbing me in last few months and letting me complete my thesis.

Special thanks to Gilbert E. Detillieux and technical team of Department of Computer Science (University Of Manitoba) for helping me with the experimental setup and providing me access to the computer resources.

Dedicated to my parents Sardarni Biran Kaur and Sardar Tikka Singh, nephew Prabhpratap Singh Sidhu, brother Jagdeep Singh Sidhu and sister-in-law Nirmal Kaur Sidhu. Without your endless support and love, i couldn't be here.

"If you want to shine like a sun, first burn like a sun." - Dr. A.P.J. Abdul Kalam

Chapter 1

Introduction

1.1 Introduction

High-Performance Computing (HPC) is used to solve complex problems in parallel for increased performance. Over the past few years, parallelization has become more challenging with the development and advancement of different computational devices based on different architectures. There are mainly two categories of architectures - CPU based homogeneous architectures and accelerator-based heterogeneous architectures which can further be classified as discrete GPUs or fused CPU and GPU multi-cores such as AMD accelerated processing units (APU) [3; 4]. Besides GPU, in recent years, field-programmable gate arrays (FPGA) also have been considered for accelerating algorithms, in particular machine learning algorithms [5] due to their power efficiency. Amazon and Microsoft offer FPGA accelerators on their Cloud based services.

One of the many challenges in these architectures is in obtaining better utilization of the resources available on these architectures through better task scheduling strategies. The goal of scheduling is to keep the execution time of a job at minimum by distributing the tasks to the processors efficiently. However, finding an optimal solution to the scheduling problem is very difficult. Therefore, researchers have been satisfied with sub-optimal solutions.

In recent years, evolutionary algorithms such as genetic algorithms (GA) [6], have been used to find sub-optimal but generally good quality solutions to the task scheduling problem. Data flow in application program can be represented by using Directed Acyclic Graph(DAG), where edges represent the communication delay between tasks and the tasks are partitioned to preserve data locality with minimum communication between tasks. To improve the quality of the solution, genetic algorithms have been combined with other strategies such as Task Duplication Strategy/Scheduling (TDS) [7].

Task duplication scheduling algorithm duplicates and allocates tasks to multiple processors to minimize the communication overhead in such a way that the required data for future tasks are readily available on processors that require them. To select eligible nodes for duplication, various techniques can be used and implemented. A few of such techniques (task duplication algorithms) duplicates only the direct predecessor tasks while other algorithms try to duplicate all possible ancestor tasks. Duplication based heuristics are useful in computing systems with high network overheads and low network bandwidths, and processors with future tasks that are sitting idle. An example of a computing system that may make use of duplication based heuristics are processors connected by wide area networks.

In scheduling tasks on high performance systems, it is important to consider the communication latency between processors together with the task dependencies in an application. In this thesis, I consider a network of CPU based machines with varying speed. I propose an efficient mapping and scheduling of tasks to processors by considering the different architectural features and communication latencies of the processors within this distributed environment. The SIPS (Serial Algorithms In Parallel System) [2] framework is used to exploit parallelism using abstract syntax trees generated directly from the source code. SIPS is a Java based framework, which supports the execution of Java programs in parallel system. This framework helps in automating the process of converting serial code for use in a parallel systems, thus reducing the overhead of writing parallel code. I have used this framework for my research since I am one of the co-authors of this framework and therefore have experience with it. The goal of SIPS is to reduce the programming overhead to implement problems in parallel system, with the minimal use of special syntax. This framework was further extended by Kumar et al.[1] with some improvements, like addition of new schedulers. In this thesis, I have extended the work further by adding the ability to locate resources scattered across the network as mentioned in Chapter 4, to implement new schedulers and to execute algorithms by using programmer defined tasks.

There are a few challenges in designing a scheduling algorithm including:

- Deciding on finding a schedule for the given tasks in such a manner that they get executed in minimal time using the available resources in the network.

- Reducing the time taken to distribute various tasks in the system (the scheduling overhead).

- Ensuring that upcoming tasks are scheduled in such a manner that they reduce the waiting time of each task due to the selection of most the suitable processing element to execute the task.

I develop a solution to solve the above challenges by using agents to map the network, a genetic algorithm to schedule the given tasks on the machines and the task duplication scheme to increase reliability and minimize the network communication between nodes. The agents collect all the required information about the resources present on the network to make the scheduling decisions. I use a master-slave approach to accomplish this step. Communication overhead, Request Processing Time, and the distance between the nodes are considered in collecting the information. After collecting information and constructing a network graph, the best scheduling strategy within minimal time is determined using the genetic algorithm. Here, performance, the architecture of the node, communication latency between nodes and nature of parallelism within task and inter-dependency between tasks are considered as parameters to the genetic algorithm. The result of the genetic algorithm is then improved further using the task duplication technique. By using task duplication, I provide the possibility of further minimizing execution time. Duplicating tasks on multiple nodes also help to reduce communication between nodes and shorten schedule lengths.

I use a master-slave approach to conduct my experiments, which is supported by the SIPS framework. The computing machines are in a distributed network environment, with varying CPU speed and/or architecture. In my implementation, only one master node schedules and distributes the tasks. The task graph is created using an abstract syntax tree. I have tested my approach to two types of problems, data parallel (task independent) and divide and conquer (task dependent). I have made a comparative analysis of my technique to other schedulers such as Chunk[8], Factoring[8], Guided Self Scheduler (GSS)[8], Trapezoid Self Scheduler (TSS)[8], Quadratic Self Scheduler (QSS)[9], and Genetic Algorithm(GA)[10] .

1.2 Thesis Organization

In Chapter 2, I have summarized all the important related work to my thesis, which assisted me in laying out all the ground work. The framework which I choose to implement my work is explained in Chapter 3. Chapter 4 explains the use of agents for network exploration for resources. In Chapter 5, I discuss the working of the genetic algorithm used for the task matching problem. Chapter 6 contains information related to Task Duplication Strategy/Scheduling (TDS), and how it affects schedule length of the schedules created by GA. Evaluation techniques and results are explained in Chapter 7 and Chapter 8, respectively. Finally, my findings are concluded in Chapter 9.

Chapter 2

Literature Review

Task scheduling has been studied on large scale distributed systems such as Cloud and Grid system ([11; 12]) and also high-performance parallel systems. Many modern day high-performance systems are heterogeneous in nature. Not all architectures are suitable for all problems. For example, accelerators are suitable for data parallel applications with structured data, while CPU based systems are more sophisticated to handle both structured and unstructured data problems. This heterogeneity adds complications to the scheduling algorithm. I have divided this section into a number of sub-sections to logically group the relevant papers.

Scheduling on high-performance parallel systems: David J Lilja [8] presented different scheduling techniques like Chunk, Factoring, Guided Self Scheduling (GSS) and Trapezoid Self Scheduling (TSS) to divide big loops into smaller chunks, thus exploiting parallelism within loops. All these schedulers used different formulas to split loops into smaller loops to keep the execution time at minimum.

Diaz et al. [9], created a new scheduler Quadratic Self Scheduler for exploiting parallelism within loops. It uses quadratic formula to split loops across different processors.

An edge-weighted directed acyclic graph (DAG) can be used to represent the data flow in a parallel program, where edges represent communication latency between different tasks and nodes represent tasks. Kwok and Ahmad [13] proposed a Dynamic Critical Path (DCP) scheduling algorithm to allocate the task graph on a parallel system. They benchmarked [14] their algorithm on a broad range of task graphs and compared the performance with other 20 (mentioned in their paper) existing algorithms. They used random graphs with predetermined optimal schedules, peer set graphs, trace-based graphs and random graphs without optimal schedules for benchmarking the algorithms. According to their results, DCP outperformed other 20 algorithms.

Kwok and Ahmad [15] also surveyed different algorithms, all with the goal of keeping the completion time at minimum and experimented using homogeneous processors for DAG allocation. They analyzed 21 such algorithms and classified these algorithms into four groups. The first group, "bounded/fixed number of processors" scheduling algorithm included algorithms that allocates the DAG to a fixed number of processors directly. The second group, an "unbounded number of cluster scheduling algorithms", allocate the DAG to an unbounded number of clusters. The third group, "task duplication based" scheduling algorithms scheduled the DAG using task duplication. The fourth category, "arbitrary processor network" scheduling algorithms allocate and map the DAG on arbitrary processor network topologies. This was one of the few earlier works in the literature when scheduling was becoming a critical topic for parallel systems. This classification helped to understand the structure of tasks and DAG and the allocation/scheduling of DAG on multiple homogeneous systems.

Oh and Ha [16] presented "a static[1] scheduling heuristic algorithm called best imaginary-

[1]Static Scheduling: selection of resources for execution before actual execution starts

level (BIL) scheduling for heterogeneous systems"[16], which used inter-processor communication and node priority to schedule different tasks on nodes. They claimed to have 20% better performance than general dynamic level scheduling which does not consider inter-processor communication as a factor to schedule tasks.

A scalable scheduling scheme called *Task Duplication Scheme* (TDS) for heterogeneous systems was introduced by Ranaweera and Agrawal [7]. This technique considered different processors have different processing power, which could result tasks could potentially have different execution times if scheduled on these processor. Prior to this technique task duplication based scheduling was mostly considered for homogeneous systems. They compared the performance of TDS with the BIL algorithm (proposed by Oh and Ha [16]) and showed that TDS is better than BIL in terms of communication-to-computation cost ratios(ccr) and makespan.

More Recently, Lima et al. [17], provided some ideas on dividing and allocating tasks on a heterogeneous system with multiple CPUs and GPUs. They studied and analyzed various scheduling strategies on these systems. They concluded that techniques such as heterogeneous earliest finish time (HEFT)[18] and data-aware work stealing [19] gave performance and speedup boost over other scheduling techniques. HEFT selects the task with highest rank/priority to be scheduled on a processor, thus using an insertion based approach on the DAG to keep the earliest finish time of the task at minimum. The whole process was divided into two stages: task prioritization and resource selection with task scheduling. Data aware work stealing (to use ideal processor for tasks waiting in queue) ensured minimal memory transfers by using meta-data information. That is, this technique kept track of data on all the nodes required for the next task. Then it allocated the next task to the node where the maximum amount of required data was available. Lima et al.[17] also introduced the use of task annotation technique that provided hints to the scheduler for making easier scheduling decisions. For example, this technique can be used to make scheduler aware of each task's resource preference (i.e. CPU or GPU).

The performance analysis of resource allocation strategies by Beaumont et al. [20] also on heterogeneous resources showed that static scheduling was better than dynamic scheduling[2] for matrix multiplication problems. Thus the same can be assumed for similar problems with bounded size. Static scheduling exploits prior knowledge of the available tasks.

Zhang et al. [21], analyzed the impact of different factors such as workload, the performance of each processing element, network performance and input-output performance on the performance and effectiveness of scheduling strategies. Their results showed that these issues played a crucial part in dynamically scheduling tasks in a heterogeneous system.

Sun et al. [11], focused on real time applications in a fault-tolerant heterogeneous system. Their results showed improvements over traditional task scheduling algorithms, by minimizing the average response time for each task, and improved the system availability without adding additional hardware cost. They considered availability and response time of each node while scheduling the task which proved a vital factor for scheduling real-time applications over the parallel system.

Frameworks: Other work in the literature has considered frameworks for harnessing the performance of the processing systems. Ahmad et al. [22], proposed a framework called, CASCH (Computer Aided Scheduling) that helped to automate scheduling on a distributed or parallel system. Sohn and Simon [23], presented JOVE, a framework to balance the load between nodes dynamically. Jove re-partitioned the load at run-time by considering the cost of data transfer between processors.

[2]Dynamic Scheduling: Allocation of resources during execution process as they become available or free

Earlier frameworks were for strictly homogeneous systems. Recently, with the advent of different type of processors, there has been tremendous interest in developing a framework or run-time system for scheduling tasks on heterogeneous multi-core architectures. StarPu by Augonnet et al. [24] presented one such run-time system. It provided multiple scheduling strategies with high-level implementations to reduce the gap between the theoretical completion time of parallel programs and actual implementation of such programs on heterogeneous multi-core architectures. StarPU[24; 25] considers load-balancing and data-locality while scheduling on CPU and GPU architectures. Further programming language supports such as OpenCL and CUDA are supported by StarPU. StarPU implemented asynchronous data transfer techniques for GPU-GPU or CPU-GPU which the authors (Augonnet et al. [24]) claim produced more performance gain than traditional scheduling techniques (such as greedy and HEFT) on heterogeneous systems.

The SIPS (Serial Algorithms In Parallel System) [1; 2] framework is focused on exploiting parallelism using Abstract Syntax Trees (ASTs) generated directly from the source code. Important information gained from ASTs is then stored into local database files. This framework helps in automating the process of converting serial code for use in parallel systems by interpreting special syntax gained from ASTs, thus reducing the overhead of writing parallel code. Kumar et al. [1] also compared SIPS with JPPF [26] and found that SIPS performs better than JPPF when scheduling matrix multiplication problem for different number of nodes. They used Matrix Multiplication as sample problem and used different schedulers (Chunk, Factoring, GSS and TSS) to compare the performance on the basis of execution time for the sample problem.

Heuristic Scheduling Techniques: Earlier work in developing heuristic-based scheduling algorithms has been primarily focused on Genetic Algorithms (GAs). Dhodhi and Ishtiaq

[27] combined GA with list scheduling heuristics. In list based static scheduling, computation time and communication time are known apriori. They claim that the combination of GA with other static scheduling techniques improved the scheduling strategy to find better sub-optimal solutions. Their proposed technique performed better than critical path/maximum immediate successor first list scheduling techniques[28].

Palis et al. [29] studied the scheduling problem as a task clustering problem. They clustered tasks to improve data locality and reduce communication overhead while scheduling the clustered tasks on the processing elements. They showed that the quality of the schedule was directly proportional to the granularity[3] of the task graph.

Bohler et al. [30], made several improvements to existing approaches, by designing and implementing scalable adaptive genetic scheduling algorithms. These algorithms reduced the schedule length for a general task graph.

Zomaya et al. [6], described a framework using GA, to provide a solution for the task scheduling problem. They showed that the combination of heuristics (to generate the initial population) with GA improves the overall performance. They also showed that GA provided more reliable and better solutions as the size of the task graphs increased because of the linear relationship between time taken to find the schedule and the size of the task graph.

Other works on GA for task scheduling have also been done for grid and Cloud computing systems (e.g. [31], [10], [32]). Some research used GPUs to parallelize the scheduling algorithms [33]. Solomon et al. [34], gave a collaborative muti-swarm PSO (Particle Swarm Optimization) task scheduling algorithm and parallelized it on a GPU. Their idea was to exploit data parallelism available in PSO algorithm to find sub-optimal solutions in minimum possible time. Their results showed a huge performance boost over sequential version

[3]The granularity of the graph is the ratio between the average communication time of the tasks and the average execution time of the tasks.

of the scheduling techniques. Sidhu et al. [35] and Beegom and Rajasree [36] used similar meta-heuristic techniques to find solutions for task scheduling and load balancing on a Cloud system.

Song and Dongarra [37] presented a new framework and techniques (including computation and communication overlap) to harness more power from existing machines. Various techniques were used to achieve load balancing among CPUs and GPUs to cultivate more processing power from existing hardware, thus resulting in reduced execution time.

Other meta-heuristic techniques such as Ant Colony Optimization (ACO) [38] inspired by ants in nature, have also been used to find better sub-optimal solutions for dynamic scheduling at run time (e.g. [39],[40]). Ant Colony Optimization as inspired from ants sends ants to traverse the network and while traversing the network they leave a trail of (depleting) pheromones behind. More ants travel the same path could result in high concentration of pheramone on a path which thus can help to identify shortest path on a network.

Hybrid Heuristic Scheduling : Over the past few years, several studies have tried to combine heuristics, for better solution quality. Gagandeep [41] showed that the scheduling techniques proposed by Bruno et al.[42] and Bażewicz et al. [43] combined with Task Duplication Scheduling (TDS) produced high-quality scheduling results, compared to other single heuristic scheduling techniques. Sri et al. [44], proposed a hybrid genetic and case based reasoning algorithm for better convergence of local optima. Their algorithm predicted the performance of processing resources available in the heterogeneous system, to improve scheduling in dynamic environments. They showed that combining case-based reasoning and GA significantly improved the solutions and generated better solutions in less time.

Benchmarking: A benchmark group to compare static scheduling algorithms was proposed by Kwok and Ahmad [14]. The diverse structure and unbiased nature of laid solid ground for the benchmarks. Scheduling scalability (SS) is a performance measure used to determine the effectiveness of scheduling algorithms in terms of solution quality, resource utilization (the number of processing elements used), and execution time.

Chapter 3

SIPS Framework

SIPS (Serial algorithms In Parallel Systems), a novice framework created by Singh et al. [2], targets parallelism using Abstract Syntax Trees (ASTs). SIPS is a Java based framework, which supports the execution of Java programs in parallel system. I have used this framework for my research since I am one of the co-authors of this framework and therefore have experience with it. As described in Chapter 2, SIPS goal is to reduce the programming overhead to implement problems in parallel systems, with the minimal use of special syntax. Later this framework was extended by Kumar et al.[1] with some improvements, like addition of new schedulers. In this thesis, I have extended the work further by adding the ability to locate resources scattered across the network as described in Chapter 4 to implement new schedulers and to execute complex problems by using programmer defined tasks.

3.1 Code Hierarchy

SIPS code is mainly divided into four parts, SIPS-lib, SIPS-Scheduler, SIPS-Node and SIPS-Run. Each of these parts share some common classes and depend on each other for the

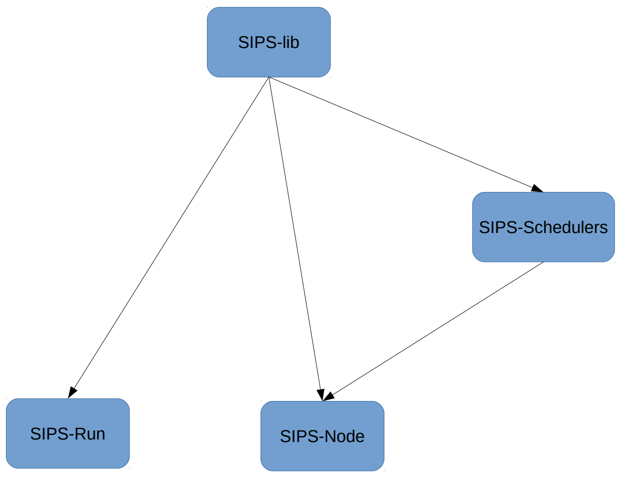

Figure 3.1: Hierarchy Of Different SIPS framework Modules

common code shared among them. Figure 3.1 shows the dependencies between these parts. As we can see from the figure, SIPS-lib is the common dependency (the root) among the other three modules. Only SIPS-Node inherits both SIPS-lib and SIPS-Schedulers. These modules are explained briefly below.

1. **SIPS-lib:** contains all the libraries essential for the execution of Jobs within SIPS. It also contains some extra utilities that allow easier development and extension of the framework.

2. **SIPS-Schedulers:** contains a set of predefined schedulers and custom data types required for their functionality.

3. **SIPS-Run** contains all the tools like the code parser and job client that are required for stage 1 of the execution process (mentioned in Algorithm 2) in Section 3.4 .

4. **SIPS-Node:** contains the code to run important services like Job Server, Task Server, Ping Server, File/Data Server, File/Data Download Server, and tools to execute Jobs and Tasks. It can act as Master and/or slave as mentioned later in Sections 3.5.2 and 3.5.3 respectively (also in Algorithm 3 and 4 in Section 3.4).

3.2 Programming Syntax

SIPS supports the execution of Java 7 programs and uses various user defined elements such as SIPS Task, Parallel For Loop and Simulate Section to exploit parallelism within the problem. These are explained below:

1. **SIPS Task** is a user defined section of code that can be executed independently from the rest of the code. Users can use the following methods to define a SIPS task.

   (a) **defineTask:** Users can define a SIPS task by calling the 'defineTask' method of the SIPS object. Method 'defineTask' accepts the unique name of the task as a parameter to avoid conflicts between multiple sections of the code.

   (b) **endTask:** To mark the end of SIPS task section, it is required to call the 'endTask' method of the SIPS object, which also accepts the same name as the parameter. All the code between these two calls is executed independently from the rest of the code. It is required for successful execution of the Job.

   (c) **setTaskDependency:** This is used to establish the dependencies between multiple tasks. It accepts task name as first parameter followed by the names of other tasks on which it depends.

   (d) **setTaskResources:** This is used to notify the scheduler about the priority of the resources (such as CPU, GPU, Storage, Memory etc.) for the mentioned task. It accepts task name as first parameter followed by the names of the resources in order of their priority.

   (e) **setTimeout:** Assigns a timeout value in the event there is a failure and the task is unable to respond at a specific mentioned time, appropriate actions can be taken by the scheduler. It accepts task name as first parameter followed by the timeout value in milliseconds.

   (f) **setDuration:** This is used to assign the weight to task. This weight can also be interpreted by schedulers in SIPS framework to assign the expected execution time to mentioned task. It accepts task name as first parameter followed by the weight value (double data type).

2. **Parallel For Loop** is the section of code that contains a for loop, which can be parallelized. Users use the following methods to define a parallel *for* loop.

   (a) **parallelFor:** marks the starting point of the section which contains the *for* loop. SIPS picks the first for loop which start after this call for parallelization. All other *for* loops (nested or trailing one's) are ignored.

   (b) **endParallelFor:** marks the end point of s Parallel For Loop section and is required for successful execution of the Job.

3. **Simulate Section** is the section of code that contains the code for the execution of simulation stage. In simulation stage, SIPS-Run comments out code except the code written in 'simulateSection' and 'endSimulateSection'. Then executes the modified code. It can contain data generation part of the Job, which can be saved by using

methods such as 'saveValue' and 'saveObject'. Then comment out code within simulate section from the original copy of code and save it for the later stages of SIPS execution of a job (discussed in Section 3.4). Thus simulate section will be excluded from the actual execution of tasks.

(a) **simulateSection:** marks the starting point of the section which can be executed in simulation stage and will be excluded from the actual execution of Tasks.

(b) **endSimulateSection:** marks the end point of a Simulate section and is required for successful execution of the Job.

3.3 Role of Abstract Syntax Tree(s) (AST(s))

SIPS converts the user-defined code to an Abstract Syntax Tree (AST). AST is a tree representation of code created by a code parser and used by the compiler to convert source code into compiled form (for example byte-code is the compiled form of Java source code).

Figure 3.2, represent an example representation of a simple for loop (from Algorithm 1.) as Abstract Syntax Tree (AST). SIPS-Run generates this kind of information from source code and saves in SQLite databases. Then during simulation stage updates the corresponding fields to 'saveValue' with the actual value of the object. Then later SIPS-Node modifies the initializer and conditional values (in case of parallelFor) with the values (Chunk sizes) generated by scheduler, during Stage 2 explained in Algorithm 3.

Algorithm 1 Example For Loop

```
1: for (int i=0; i<=10; i++) {
2:   //Body
3: }
```

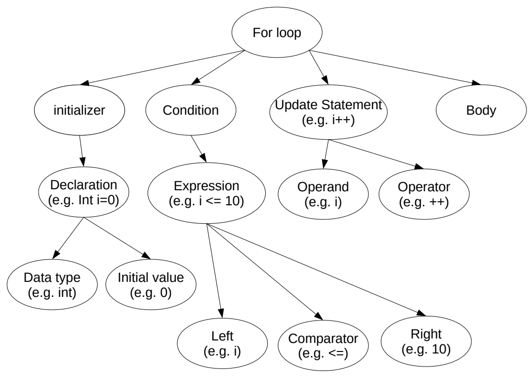

Figure 3.2: Example of an AST for a simple for loop

In case of SIPSTask, SIPS-Node comment out code except the task it is scheduling, to do so it extracts the information from the databases generated from ASTs (created during Stage 1).

3.4 Execution of Job

SIPS executes a Job[1] in three stages. The execution of stages 1 and 3 run separately but overlap with the execution of stage 2. In the first stage (also represented in Algorithm 2), SIPS-Run is used to perform all the operations. SIPS-Run reads the manifest file, which a file in JSON format to read special values like the number of nodes to use for the execution of the job, job name, Master Node's address (ip/hostname:port) etc. (See sample manifest file in appendix's section A.1). Then it contacts the master node with a request to get job token. Then it parses the source code and gathers all the information related to user defined tasks (SIPSTask), parallel for loop and data interacting within these elements. Information

[1]Job is different from Task in this thesis. Job is a complete process that constitutes one or many tasks(SIPS Task)/parallel loops, whereas a task is the smallest portion of the Job that can be executed in parallel with other tasks.

gathered in the first stage is stored for processing in the second stage. SIPS-Run then execute the *Simulate Stage*. SIPS-Run is used in the first stage to perform all these actions and interact with the master side of the framework to generate a Job token, data transfer (databases which contains ASTs, objects generated during simulation stage etc.) and start the Job.

In the second stage (represented in Algorithm 3), (SIPS framework use SIPS-Node to perform all these operations) the Master Node use agents to gather information about other nodes available on the network, grants job token, accepts the Job request and then parses the information, selects the defined scheduler for the task and distributes the smaller tasks to nodes chosen by the scheduler, after making modifications to the abstract syntax tree(s). In Algorithm 3, submitter is the host which is using SIPS-Run to execute the Stage 1 of the job.

In the third stage (represented in Algorithm 4), chosen nodes execute the smaller tasks and send back the results to the Master Node.

Algorithm 2 SIPS Framework- Stage 1 (SIPS-Run)

```
 1: procedure SipsRun
 2:     Start
 3:     Read manifest file.
 4:     Contact master node and requests a job token.
 5:     Read Source Code.
 6:     Generate Abstract Syntax Trees (ASTs).
 7:     Store ASTs in Database Files.
 8:     Comment out code except simulate section from source files.
 9:     Start Simulation Stage.
10:     Analyze data dependence in the code, save objects and values in simulate section.
11:     End Simulation Stage.
12:     UnComment code and Comment out simulate section.
13:     Upload all the files (source code, manifest, databases, objects etc.) to Master Node.
14:     Send Start Signal with Job Token.
15:     EXIT
```

Algorithm 3 SIPS Framework- Stage 2 (SIPS-Node Master)

```
1: procedure SIPS-Master
2:     START
3:     On New Thread Use Agents to locate resources on network.  ▷ explained in Chapter 4
4:     On New Thread Start Service to handle file/data requests.
5:     while ($STOP = FALSE$) do
6:         Accepts new job request from submitter (SIPS-Run), generate a job token and send job token to submitter (SIPS-Run).
7:         Accepts all the files (source code, manifest, databases, objects etc.) and move them to appropriate directory (assigned to that job).
8:         Wait for Start Signal from submitter (SIPS-Run).
9:         Read Manifest File.
10:        $Result = Scheduler(L_n, Data)$.  ▷ where it accepts list of live nodes and type of data to schedule, like Parallel For and SIPS Task, and returns data paired with nodes
11:        Distribute the $Data$ to paired nodes.
12:        On New Thread Wait For Results.
13:        On New Threads Serve Results.  ▷ Uses Multiple Threads
14:        On New Thread Receiving Last result, calculate total time, average cache hit-miss ratio, upload speed, upload data, cached data and store in local data warehouse .
15:    end while
16:    EXIT
```

Algorithm 4 SIPS Framework- Stage 3 (SIPS-Node Slave)

```
1: procedure SIPS-Slave
2:     START
3:     Benchmark the Resources (HDD,CPU etc)
4:     On New Thread Start Service Respond to Agents.  ▷ See Chapter 4 for more information.
5:     On New Thread Start Service to handle data dependencies.  ▷ This service transfers objects, results and files upon receiving the request from the tasks this node currently executing.
6:     while ($STOP$ =FALSE) do
7:         On New Thread Accepts Job Data.
8:         On New Threads Solve File/Data Dependencies.  ▷ Instead directly contacting nodes to fulifill data dependencies, it relies on the service which queues all the data request mention in early steps of this algorithm.
9:         On New Thread Start Execution of the job.
10:        Send Results back to master node.
11:    end while
12:    EXIT
```

3.5 Execution Modules

SIPS has three modules (on the basis of roles) which play different roles during execution of a Job.

3.5.1 SIPS-Run

SIPS-Run handles the execution of the first stage (see Algorithm 2). It mainly consists of the following elements.

1. **Code Parser:** converts the source code into abstract syntax tree (AST), extracts the required information (such as intial value, conditional value, update value, variable name etc. in case of for loop) and stores it in the SQLite databases. (Also explained in Section 3.3)

2. **Code Simulator:** uses information generated by Code parser, comments out the code except Simulate section, executes the code to analyze data dependencies, saves data/values required during execution stage, and uncomments the source code and comments out Simulate section.

3. **File/Data Uploader:** sends all the data/information generated by code simulator, and all the other files which are required to execute the Job to the master.

4. **Job Client:** requests master to provide a Job token and handles additional responsibilities of sending the start signal and checking the Job status.

3.5.2 Master

The **Master side** of the framework runs essential services (like Job server, file/data server, API server etc.) on multiple threads. It handles execution of stage 2 (see Algorithm 3). Components used by the Master side are described below:

1. **Job Server** handles the multiple stages of a Job, from Job token creation, receiving all the necessary files and data from submitter, to starting the Job.

2. **File Server** handles the file requests made by tasks before and during execution. All the data related to a particular job is transferred through master node, even the nodes which have completed the execution of the tasks assigned to them, send results to master node and nodes who want to access results to complete the execution of tasks assigned to them, send their requests to master node's file server.

3. **API (Application Programming Interface) Server** provides the access to API(s) which can be used to retrieve data from SIPS framework and can also be used to manipulate the settings and services of the SIPS framework.

3.5.3 Slave

The **Slave side** of the framework runs essential services (like task server, file download server, ping server etc.) on multiple threads. It handles execution of stage 3 (see Algorithm 4). Components used by the Slave side are described below:

1. **Task Server** handles the tasks submitted by the Master Node, submits the requests to file download server (see below) to download all the necessary data required to start the execution of the tasks and starts the execution of tasks.

2. **PING Server** handles the ping request made by other nodes and replies back with all the information related to current node like CPU Model, CPU architecture, Number of CPU Cores, Hard drive size, Hard drive free space and benchmark[2] results of these resources.

3. **File Download Server** handles the file/data requests made by tasks before and during execution. Then it forwards those requests to Master Node's file server. It plays the role of a common download queue for all the tasks to keep the CPU and the Network load at minimum levels.

3.6 Supported Layouts

The SIPS framework supports the execution of tasks even via complex setup of nodes such as Peer 2 Peer (P2P) (as shown in Figure 3.3) and Master Slave (as shown in Figure 3.4) Architectures. In P2P layout two (or more) master nodes can submit tasks to each other for execution. As it is also represented in Figure 3.3, Node A and Node B are execution two Tasks X and Y, where for Task X, Node A acts as a Master and Node B acts as a slave, and for Task Y Node B is acting as Master and Node A is acting as slave. For Master-Slave layout (as shown in Figure 3.4), Node 1 acts as a Master node to other nodes which act as Slave nodes and are connected via the network.

[2]SIPS use JDiskMark library to benchmark hard drive and SciMark library to benchmark CPU

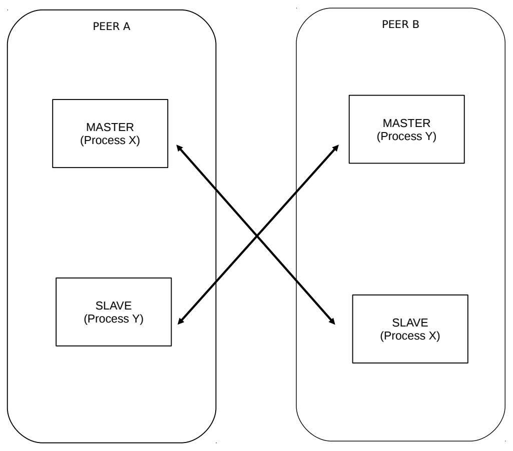

Figure 3.3: SIPS P2P setup, Image by Kumar et al. [1], [2]

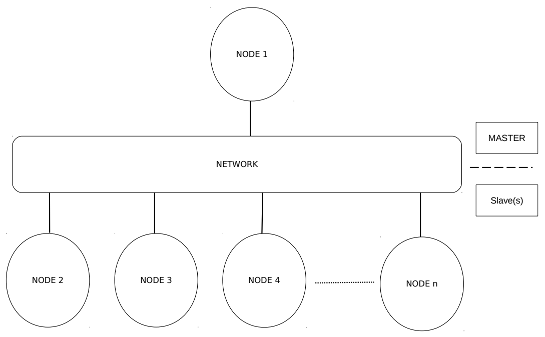

Figure 3.4: SIPS Master Slave setup, Image by Kumar et al. [1], [2]

3.7 Cache Exploitation

SIPS nodes use a local cache on each machine's hard drive to reduce the communication between nodes. As multiple tasks from the same job can request for the same file/(data/objects), using a disk cache can help in minimizing network traffic, CPU load and maximizing data reuse. The file download service mentioned in this chapter in section 3.5.3 takes advantage of local cache storage to prevent duplication of file/data requests.

3.7.1 Cache hierarchy

Local cache storage is organized the cache by node *uuids*[3]. These directories contain sub directories named using Job token and then the relative path to file on the file server (also

[3]uuid stands for Universally Unique Identifier, SIPS use two 128-bit uuids combined in one string for node identification

shown below, as 3.1).

$$cache/ < node - uuid > / < job - token > / < relative - path - to - file > \tag{3.1}$$

3.7.2 Cache Implementation and Exception

SIPS uses *sha checksum* of files to check the validity of local files stored in the cache by comparing it with the *checksum* received from the server. The only exception in *checksum* matching is to results which belong to the same job and tasks executing on the same node. This helps in reducing the execution time of job and minimize the wait time of the successor or dependent tasks, which do not need to wait to receive *checksums* from the master node which hasn't even received the result yet. To avoid the waiting time that can occur due to transfer of results from slave to master and master to slave (here slave is the same node which is sending results to master), we store a copy of the results in the cache and transfer the results later to the master node, to prevent the other tasks from getting cache misses. Other nodes that are executing dependent successor tasks need to wait for the master node to receive those results and then access those results from the master node to complete their execution.

Chapter 4

Using Agents to Locate Resources

In this chapter, I discuss how i have implemented agents to explore resources which are available over the network. Throughout this chapter I have used COH for Communication overhead, RPT for Request Processing Time. The technique i used is inspired from techniques such as ant colony optimization and particle swarm optimization mentioned in Chapter 2.

To perform the resource allocation task, I have used agents to collect all the required information to make the scheduling decisions. At the start, the master processor has no information about the computing devices (nodes), and these nodes are scattered over the network in a non-dedicated environment. The agent exploration technique is illustrated by using Fig. 4.1 with five nodes, where, $N1$ is the master processor. All the participating nodes, upon start-up, load the list of IP addresses (eg. 192.168.0.111) and network addresses (eg. 192.168.0.0/24). From this list of network addresses and by using the current sub-net address of the node, it generates another list of IP addresses to scan (as shown in Algorithms 5 and 8, which use Algorithm 6 to perform this task). In Fig 4.2 every node becomes aware of the IP addresses it has to scan. As we can see from Fig. 4.3, all nodes send agents to adjacent nodes in the network to collect data about the participating nodes. At this stage, all the nodes send a unique hash *id* (representing themselves) to the agents so that the nodes are aware of the nodes participating in the network. On the receiving end, receivers use these unique id's and IP address (of the sender) to restrict/filter blacklisted[1] nodes. The procedure of the sender side node is represented in Algorithms 5, 6 and 7. Then sender processor waits for the agent to return back with the information as illustrated in Algorithm 6.

When an agent reaches a node, N_i, it collects all the information (such as Hostname of machine, Information of Adjacent Nodes, Information of Non-Adjacent Nodes, Type of CPU, Total Number of CPU cores, Storage Device, *currently* maximum available memory/RAM, average CPU load etc.) and returns the information back to the sender processor (Fig. 4.4), as described in Algorithm 8. In addition to the current node's information, the agent also sends a list of adjacent and non-adjacent nodes separately with minimum distance from the current node (described in a later part of this chapter). Unique *id* (uuid) generated by each node is used to communicate within system. The sender node also tracks the time taken to send the agent and the time to receive the agent back with the information. This communication delay indicates approximately the latency between the nodes and the time it may take to send a message to that node in the network. This communication overhead is calculated using Equation 4.1. Both sides keep track of the network delay and keep sending latest values during future exchanges of information between the nodes and the master processor. The communication delay is calculated by tracking the execution time of different requests, so there is no need for a globally synchronized clock.

To send back the information to the sender node, the adjacent nodes also send agents

[1]SIPS maintains a list of UUIDs and IPs, which are not allowed to access any functionality of the system. user with correct permission level can add or remove values from this list by using SIPS api.

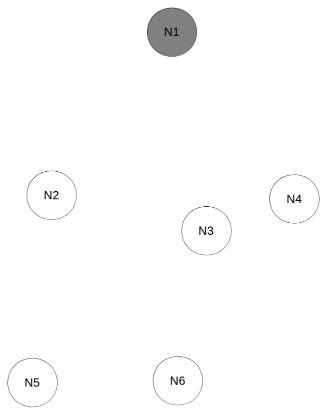

Figure 4.1: Isolated Nodes in the Network.

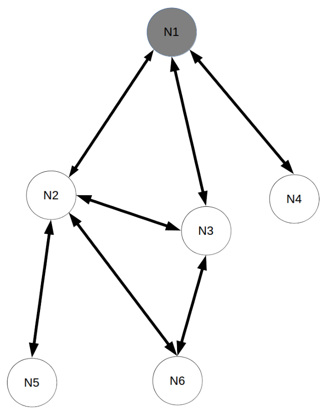

Figure 4.2: Network Connection Established.

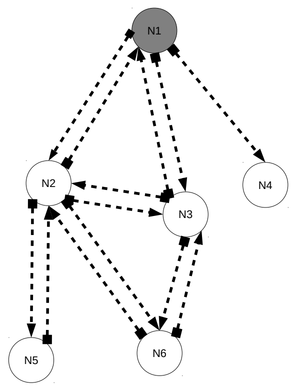

Figure 4.3: Agents exploring adjacent Nodes.

along to their adjacent nodes (in parallel) to explore the network, as can be seen in Fig. 4.4, and also represented in Algorithm 8 and 6.

I have used the following formula to calculate the communication overhead (COH) between two nodes, N_i and N_j:

$$COH(N_i, N_{adj}) = TotalTime(N_i) - (ExecutionTime(N_{adj})) \tag{4.1}$$

where node N_i is the sender/master node and node N_{adj} can be the receiver/slave node. TotalTime(N_i) is the time recorded by node N_i from sending agent to receiving reply. ExecutionTime(N_{adj}) is the time recorded by node N_{adj} from start of the request processing to the end of request processing(just before sending reply).

This value can be divided by 2 to get the approximate value for one sided communication delay. Agents also send a list containing information about adjacent nodes and a list containing information of non-adjacent nodes to the node N_i. This allows each participating node to construct the connected graph for the whole network. Nodes also keep scanning the

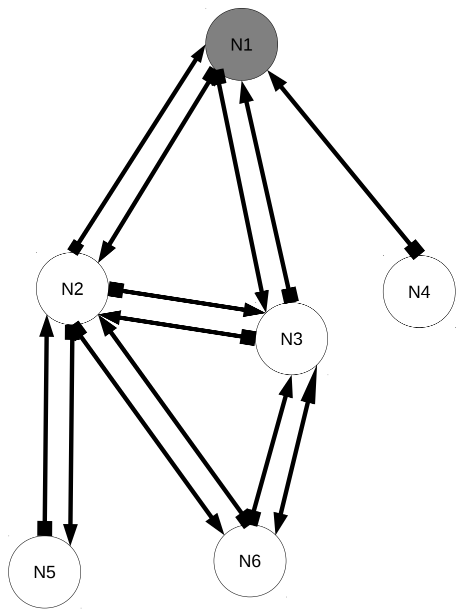

Figure 4.4: Agents from Adjacent Nodes Returning With Information.

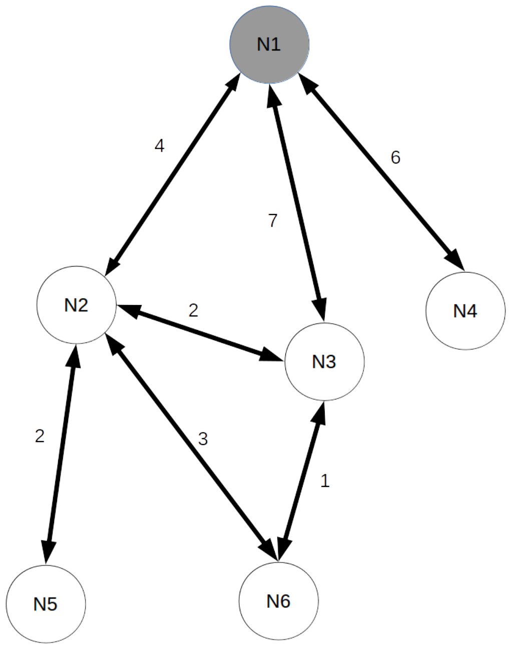

Figure 4.5: Explored Network with now known Communication Overhead between Nodes.

network to check for the availability of new nodes or to detect the failure of already available nodes, after a finite interval. This repitation of this process after a finite interval also helps Agents also keep the information up to date.

Algorithm 5 Master

```
1: procedure MASTER
2:     Boot.
3:     Generate Unique Hash ID and assign to itself.
4:     New Thread($MapNetwork()$).                    ▷ Algorithm 6
```

As mentioned, using Algorithm 7, the agent also builds two tables, the Adjacent table and the Non-Adjacent table which contain the information of adjacent and non adjacent nodes, respectively. After receiving the information via the agents, the nodes also iterate

Algorithm 6 Network Mapping Algorithm

```
 1: GLOBAL AdjNodeTable,NonAdjNodeTable,RoutingTable
 2: procedure MapNetwork
 3:     $List_A$ ← Load list of ip addresses and network addresses.
 4:     $List_B$ ← Load list network addresses.
 5:     $List_C$ ← Generate List of ip addresses on current subnet and for each element of
        $List_B$.
 6:     $Nodes$ ← Concatenate ( $List_A$ and $List_C$ ).
 7:     while true do                                            ▷ Infinite Loop
 8:         while ($Node \leftarrow Nodes.Next()$) $\neq null$ do ▷ For every active participating node
 9:             New Thread(CollectInformation(Node)) ▷ Create a separate thread to collect
                information of particular node, (using Algorithm 7.)
10:         end while
11:         sleep                  ▷ For a finite interval to reduce unnecessary load
12:     end while
```

through the adjacent and non-adjacent tables received from the adjacent node. They check if the Adjacent table has a value under the same node id. If yes, then they compare the time stamps and update the value if the received information is the latest. To perform this operation, we compare values using *checkedAgo* method. If Adjacent table doesn't contain the values under the given node id, then it performs another check to see if the node is adjacent or not. If the node is adjacent to the current node we again can insert the new value in the adjacent node, other wise we can look up in the Non-Adjacent table and perform insert or update operation using the same checks we have performed on Adjacent table. For the Non-Adjacent nodes, we maintain another table where we keep the information related to Non-Adjacent nodes.

Algorithm 7 Collect Information of Node

```
procedure CollectInformation(Node $N_{adj}$)
    $StartTime \leftarrow CurrentTimeStamp$
    Send Agent to $N_{adj}$ accompanied by Current Node's UUID
    Wait For Response from $N_{adj}$
    Information($N_{adj}$) = Receive Agent from $N_{adj}$ with Collected Information
    $EndTime \leftarrow CurrentTimeStamp$ AND $RPT \leftarrow Information(N_{adj})$.RPT
    $COH(N_{adj}) \leftarrow$ ( $EndTime - StartTime$ ) - ($RPT$)
    $Information(N_{adj})$.COH $\leftarrow COH(N_{adj})$
    $Information(N_{adj})$.CheckedOn $\leftarrow EndTime$)    ▷ CheckedOn can be used to calculate CheckedAgo by substracting current time stamp from CheckedOn
    if AdjNodeTable.has($N_{adj}$) then
        Update AdjNodeTable($N_{adj}$) $\leftarrow Information(N_{adj})$
        Update RoutingTable($N_{adj}$) with Hop($N_{adj}$) $\leftarrow Information(N_{adj})$.COH
    else
        Insert AdjNodeTable($N_{adj}$) $\leftarrow Information(N_{adj})$
        Insert RoutingTable($N_{adj}$) with Hop($N_{adj}$) $\leftarrow Information(N_{adj})$.COH
    end if
    while ($Val \leftarrow Information(N_{adj}$.AdjNodeTable.next()) $\neq NULL$ OR ($Val \leftarrow Information(N_{adj}$.NonAdjNodeTable.next()) $\neq NULL$ do
        Update/Insert RoutingTable($Val$) with Hop($N_{adj}$) $\leftarrow Information(Val)$.COH
        if AdjNodeTable.has($Val$) then
            if $Val$.CheckedAgo < AdjNodeTable.$Val$.CheckedAgo then
                Update AdjNodeTable($Val$) $\leftarrow Information(Val)$
            end if
        else
            if $Val$ is adjacent then
                Insert AdjNodeTable($Val$) $\leftarrow Information(Val)$
            else
                Add new Hop to $Val$ if doesn't have with ($N_{adj}$).id
                ($Val$).COH $\leftarrow$ ($Val$.COH + ($N_{adj}$).COH)
                if NonAdjNodeTable.has($Val$) then
                    if $Val$.CheckedAgo < NonAdjNodeTable.$Val$.CheckedAgo then
                        Update NonAdjNodeTable($Val$) $\leftarrow Information(Val)$
                    end if
                else
                    Insert NonAdjNodeTable($Val$) $\leftarrow Information(Val)$
                end if
            end if
        end if
    end while
EXIT
```

Algorithm 8 Slave

```
1: procedure SLAVE
2:     Boot
3:     Generate Unique Hash ID and assign to itself.
4:     New Thread($MapNetwork()$).
5:     while true do                    ▷ Infinite Loop to accept new Connections
6:         Check Sender's IP and/or id against blacklist
7:         if (Sender is not Blacklisted) $AND$ (Packet is Agent) then
8:             Receive Agent From Master
9:             $StartTime$ ←CurrentTimeStamp
10:            Add Information about the node to Agent
11:            Add Adjacent Nodes Table and Non Adjacent Table to Agent with minimum
    distance of each node (from current node) by using Routing table
12:            $EndTime$ ←CurrentTimeStamp
13:            $Information$.RPT ← ($StartTime - EndTime$ )
14:            Send Packet(Information,Master)
15:        end if
16:    end while
```

	N1	N2	N3	N4	N5	N6
N1	0	0	0	0	0	0
N2	0	0	0	0	0	0
N3	0	0	0	0	0	1
N4	0	0	0	0	0	0
N5	0	0	0	0	0	0
N6	0	0	1	0	0	0

Table 4.1: Example Network Representation on Step 1

	N1	N2	N3	N4	N5	N6
N1	0	0	0	0	0	0
N2	0	0	2	0	2	[3]
N3	0	2	0	0	[4]	1
N4	0	0	0	0	0	0
N5	0	2	[4]	0	0	[5]
N6	0	[3]	1	0	[5]	0

Table 4.2: Example Network Representation on Step 2

	N1	N2	N3	N4	N5	N6
N1	0	0	0	0	0	0
N2	0	0	2	0	2	[3,3]
N3	0	2	0	0	[4]	[1,5]
N4	0	0	0	0	0	0
N5	0	2	[4]	0	0	[5]
N6	0	[3,3]	[1,5]	0	[5]	0

Table 4.3: Example Network Representation on Step 3

	N1	N2	N3	N4	N5	N6
N1	0	4	[6]	0	[6]	[7]
N2	4	0	2	0	2	[3,3]
N3	[6]	2	0	0	[4]	[1,5]
N4	0	0	0	0	0	0
N5	[6]	2	[4]	0	0	[5]
N6	[7]	[3,3]	[1,5]	0	[5]	0

Table 4.4: Example Network Representation on Step 4

	N1	N2	N3	N4	N5	N6
N1	0	4	[6]	6	[6]	[7]
N2	4	0	2	[10]	2	[3,3]
N3	[6]	2	0	[12]	[4]	[1,5]
N4	6	[10]	[12]	0	[12]	[13]
N5	[6]	2	[4]	[12]	0	[5]
N6	[7]	[3,3]	[1,5]	[13]	[5]	0

Table 4.5: Example Network Representation on Step 5

	N1	N2	N3	N4	N5	N6
N1	0	[4,9]	[6,7]	6	[6,11]	[7,8]
N2	[4,9]	0	2	[10]	2	[3,3]
N3	[6,7]	2	0	[12]	[4]	[1,5]
N4	6	[10]	[12]	0	[12]	[13]
N5	[6,11]	2	[4]	[12]	0	[5]
N6	[7,8]	[3,3]	[1,5]	[13]	[5]	0

Table 4.6: Example Network Representation on Step 6

Destination	Hop	Distance
N1	N1	0
N2	N2	4
	N3	9
N3	N2	6
	N3	7
N4	N4	6
N5	N2	6
	N3	11
N6	N2	7
	N3	8

Table 4.7: Snapshot of Node N1ś routing table

To keep routing simple, we also maintain a routing table[2] (as shown in the example in Table 4.7). This routing table contains the primary key as the id of the node which is adjacent or non adjacent to the current node. In the second column we keep the id of the adjacent node which can relay information. The third column contains the communication overhead (COH) from current node. All the information in the second and third columns are sorted in ascending order of distance from current node. If a node is adjacent, then the destination and hop id is the same and the distance is the communication overhead we observed for the adjacent node. However, for non adjacent nodes, to keep track of the communication overhead in case we need to relay information through traversed nodes, we add the communication overhead observed from the adjacent node to the communication overhead observed for the adjacent node, and store this in the routing table with the adjacent node's id as Hop. When choosing the non adjacent nodes for task execution, when adjacent node with minimum communication overhead is selected to relay information between the master node and selected adjacent node.

Tables 4.1 though 4.6, represent the connectivity of all nodes, where discovery of each node is considered as a step. There is a co-relation between each step and distance between two nodes in the network. We assume all the nodes boot up at the same time and send agents (in parallel) to explore the network at the same time. We also assume the Request Processing Time (RPT) is constant, and the distance between nodes varies. Thus two nodes with minimum distance between them gain information faster about each other than the nodes with higher distance between them.

[2]Worst-case Space Complexity of such routing table will be O(Number of Adjacent Nodes x Number of Total Nodes in the system). Worst Case time complexity of such tables can be O(1+1), if Hashmaps are being used to maintain these tables. SIPS use ConcurrentHashMap to maintain this table.

Chapter 5

Genetic Algorithm for Scheduling

The agents discussed in Chapter 4 collect the information from the network and construct a network graph. The next step is to find the best scheduling strategy within the smallest reasonable time. I achieve this by using a genetic algorithm (GA). I consider performance, the architecture of the node, communication latency between nodes and nature of parallelism within each task and inter-dependencies between tasks as parameters to the GA. The scheduling also depends on certain *methods* (also mentioned in Chapter 3) defined by the programmer/end-user who submit tasks to the system for execution. Below is a sample of a *method* that can be used to set resource priority for a task. Here, STORAGE represents resources that can be used as storage devices (like a Hard disk drive(HDD) or Solid State Drive (SSD) etc.), CPU represents processing resource (like Processors of various architectures i386, x86-64, ARM etc.).

```
sips.setResources(STORAGE,CPU)
```

In the above example, the programmer has set STORAGE as his/her highest priority. If STORAGE is not available the *method* also indicates to use a node with CPU resources.

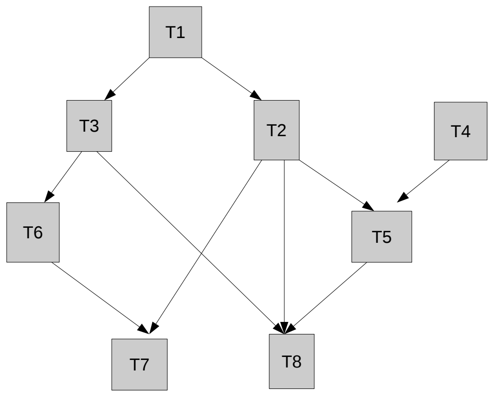

Figure 5.1: Example of Directed Acyclic Graph(DAG) of Tasks.

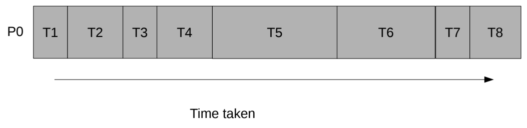

Figure 5.2: Task Scheduling on Single Processor.

In my implementation, I use the SIPS framework [2] introdued in Chapter 3. SIPS converts the user-defined code to an Abstract Syntax Tree (AST). AST is a tree representation of code created by a code parser and used by the compiler to convert source code into compiled form (for example byte-code is the compiled form of Java source code). With the help of *methods*, I split the code into different tasks. The master node maintains a list of resources about different nodes, based on the information collected using the agents (explained in Chapter 4). The scheduler constructs a directed acyclic graph of the program code. Fig. 5.1 shows an example of 8 interdependent tasks. In the following example, I assume two processors P0 and P1 are to be allocated the tasks mentioned above (or vice versa).

In any genetic algorithm, there are four important operations: generation, selection, crossover and mutation. I define a mapping of information to “chromosomes” and use this technique to finalize a scheduling strategy to allocate resources to different tasks.

1. **Generation:** Consider the tasks (T_1,T_2,T_3...T_n). The length of each chromosome

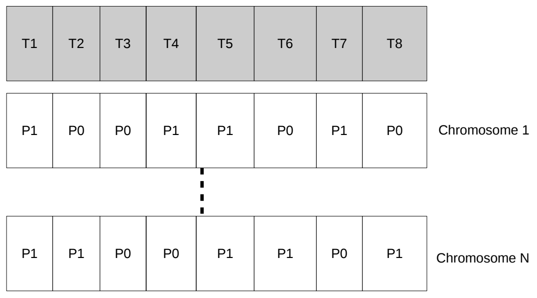

Figure 5.3: Chromosomes generated using Roulette's Wheel Method

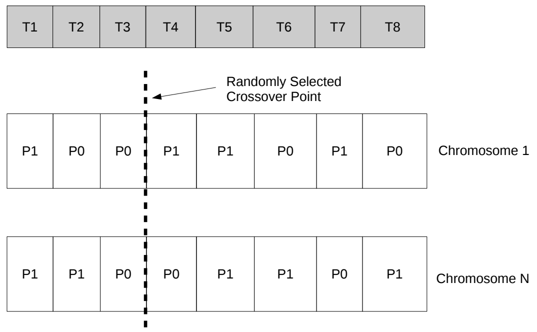

Figure 5.4: Random Selection of Crossover point Genetic Algorithm.

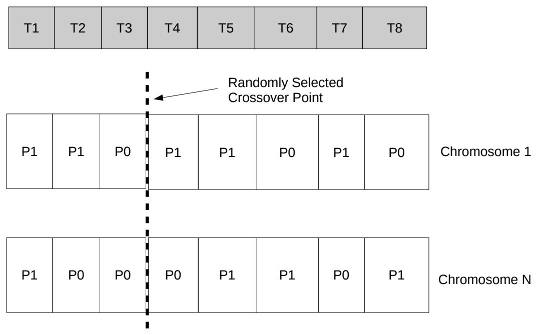

Figure 5.5: After Crossover.

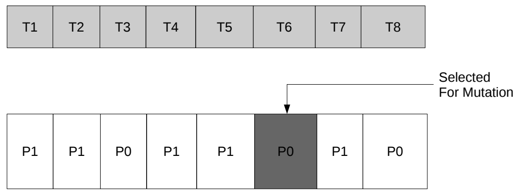

Figure 5.6: Element Selection For Mutation.

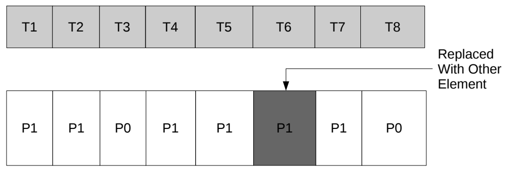

Figure 5.7: Chromosome After mutation.

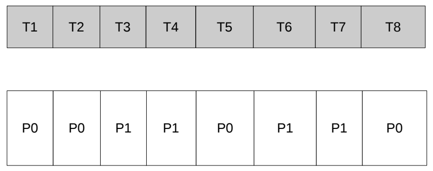

Figure 5.8: Final Chromosome.

corresponds to the total number of tasks (Fig. 5.3). Each task is allocated to a processor. A resource is selected randomly using the roulette wheel[1] method [10; 45] from the list of available resources which is built according to the resource priority

[1]Roulette wheel method selects a random element from the provided list of inputs and returns as output.

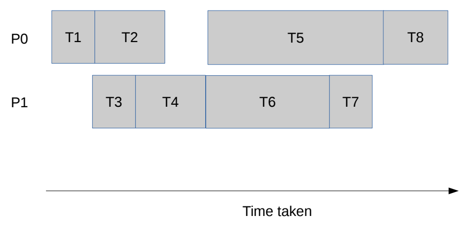

Figure 5.9: Schedule length for Example.

defined for the task. . Other factors of resource selection are communication delay between resources and performance of resources. The scheduler maps the directed acyclic graph to the network according to task interdependence and communication overhead between nodes.

2. **Selection:** In this step, by using the roulette wheel method [10] two chromosomes are selected randomly for the next step.

3. **Crossover:** In this step, the GA performs the crossover operation offer selecting a random crossover point. After crossover, the best chromosome is selected for next step on the basis of expected finish time. As we can see from Fig. 5.4 and Fig. 5.5 subparts of Chromosome 1 and Chromosome N get interchanged as the result of the Crossover process.

4. **Mutation:** In this step an element is selected randomly and replaced with the resource with better performance to improve the schedule length (Fig. 5.6 and Fig. 5.7). The result is then selected for next iteration of steps[2].

The termination condition for the GA processing is either the number of iterations or no improvement (reduction) in the schedule length. The resultant chromosome for the above

[2]Generation step is executed only once.

example after few iterations is shown in Fig. 5.8 and the final schedule is shown in Fig. 5.9.

$$SL(Chromosome_n) = C(T_1)/CPT(P_{sel})+C(T_2)/CPT(P_{sel})+...+C(T_N)/CPT(P_{sel}) \quad (5.1)$$

From equation 5.1, to compute schedule length (SL) GA assume processor's (P_{sel}) performance is to handle computations per time unit (CPT), and for each task length need C computations, so that gives an estimate execution time for each task by dividing C with CPT as C/CPT. Then adding up all the estimate completion times for each task, that gives an estimate finish for a chromosome.

In this chapter, I have included the algorithm used for finding the optimal schedule length before Task Duplication Stage (Chapter 6). As mentioned in Algorithm 9, initially our algorithm takes input from a manifest file for variables like, maximum number of nodes (need for this job), maximum number of generations (for Genetic Algorithm) and maximum population (number of chromosomes). Then we sort the processors according to available task slots, then average load observed on CPU, maximum tasks it can handle in parallel and the distance from current node. A sublist of processors with size of *MaxNodes* is created, only if list of processors is greater than the *MaxNodes* number. This helps in selecting best available processors for our Job. Then GA sorts tasks according to their dependencies, so that independent tasks can be scheduled before dependant tasks. GA creates n number (equal to MaxPopulation) chromosomes and assign random[3] processors using Roulette's wheel method. After generating the initial population, GA select one chromosome randomly at the beginning of the crossover step and one chromosome randomly for each generation.

[3]In my implementation i have used Java's Random class so that we get different non-repeating random number each time.

GA also select a crossover point randomly for the length of chromosome, on that cross over point chromosomes exchange their elements. After cross-over GA re-calculate schedule lengths according to new allocated processors. GA selects the best chromosome among the selected two chromosomes on the basis of minimum schedule length, for the mutation step.

In the mutation step, GA selects a random element of the best chromosome and assign a better processor (also randomly selected among the best, if any available) which may for example have a low load distribution than the currently allocated processor. Then again, after re-calculating schedule lengths of the best chromosome, we repeat this in finding better chromosomes for the next generation until we satisfy the *MaxGenerations* condition in the for loop, as shown in Algorithm 9. As a result,GA returns best possible chromosome so that all the tasks can be distributed to the assigned processors.

Algorithm 9 Genetic Algorithm

```
1: procedure GENETIC ALGORITHM(Processors, Tasks)
2:     MaxNodes ← JobManifest.MaxNodes
3:     MaxGenerations ← JobManifest.MaxGenerations
4:     MaxPopulation ← JobManifest.MaxPopulation
5:     Sort list of processors according to Available Task Slots, CPU Average Load, Maximum Tasks it can execute in parallel, CPU Score and then distance from the current node.
6:     Processors ← sublist of processors of size MaxNodes.
7:     Sort Tasks According to their dependencies.
8:     Create an Empty List for Chromosomes.
9:     for i $\leq$ MaxPopulation do
10:        Sort Processors by choosing a random sorting factor from Available Task Slots, CPU Average Load, Maximum Tasks it can execute in parallel, CPU Score and then distance from the current node.
11:        for each Task in Tasks do
12:            Select a random processor from Processors and assign to Task    ▷ Using Roulette Wheel Method
13:        end for
14:        Create a Chromosome with Tasks as it's elements
15:        Add Chromosome to Chromosomes list
16:    end for
17:    BestChromosome ← new Chromosome
18:    RandomChromosome ← Randomly Choose a Chromosome from Chromosomes
19:    for i $\leq$ MaxGenerations do
20:        RandomChromosome2 ← Randomly Choose a Chromosome from Chromosomes
21:        Randomly Choose a crossover point
22:        Crossover RandomChromosome and RandomChromosome2 at crossover point
23:        Re-calculate Schedule Lengths
24:        BestChromosome ← MinScheduleLength (RandomChromosome, RandomChromosome2 )
25:        Randomly Choose a mutation element
26:        Replace processor of randomly chosen mutation element with a randomly chosen better processor    ▷ On the basis of CPU score
27:        Re-calibrate Schedule Lengths
28:        RandomChromosome ← BestChromosome
29:    end for
30:     return BestChromosome
```

Chapter 6

Task Duplication Strategy for Scheduling

The result of the genetic algorithm can be improved further using the task duplication technique [7]. By using task duplication strategy (TDS), I can try to further minimize execution time. Duplicating tasks on multiple nodes will also help to reduce communication between nodes and shorten schedule lengths.

The TDS technique checks if it is better to duplicate a task on another node or leave the tasks where they are and transfer data between nodes as needed by the *parallel program*. First, I calculate the total calculations($TOT_{calculations}$)[1] required to finish all the tasks. Then by dividing total calculations($TOT_{calculations}$) by the performance of the processor (i.e. the CPU score we obtain from benchmarks mentioned in Chapter 3), we can set the priority factor (using 'setResources' method mentioned in Chapter 3) for the resources. This will be helpful to determine if the candidate task (i.e. task to be duplicated) will take less time on

[1]In many cases, this will presumably be an *estimate* since runtime factors may determine the number of calculations needed.

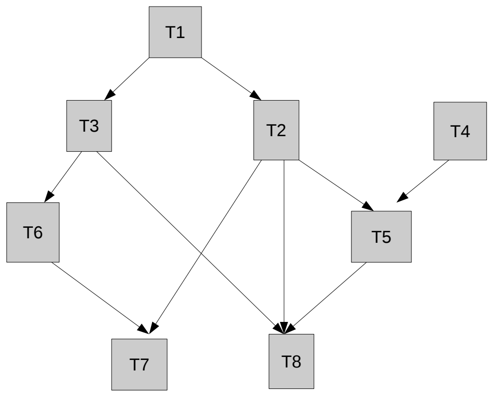

Figure 6.1: Example of Directed Acyclic Graph(DAG) of Tasks.

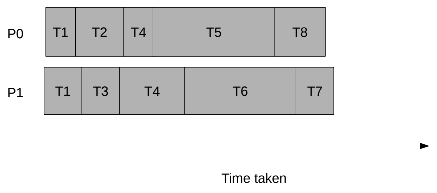

Figure 6.2: Improved Schedule length After task Duplication.

selected processor (i.e. new processor where the duplicate task will execute). To check if the duplicating task will be beneficial, this strategy compares the estimated time to transfer required data from one processor to another to the time reduced by duplicating the task on the selected processor (see Algorithm 12). During this process the dependencies of the candidate task, predecessor task of the candidate tasks on the current processor and their estimated start and finish times are considered so that the task to be duplicated fits in free slots available inn the schedule length on the target machine. After applying this technique in the example above, the final schedule length obtained is shown in Fig. 6.2, which is an improvement over the schedule length (in Fig. 5.8) generated by the genetic algorithm alone. In Fig 6.2 we can observe that T4 has different execution time because of the different CPU speed, and is duplicated which eliminates the need of data transfer between the processors and reduce the wait time for Task T5.

In Algorithm 10, we take the output of the GA as input for TDS. In Algorithm 11, for

Algorithm 10 Task Duplication Scheduling/Strategy(TDS)

```
1: procedure TDS($Processors$, $Tasks$)
2:     create new empty list for tasks as TaskList
3:     BestChromosome ← GeneticAlgorithm($Processors$, $Tasks$).
4:     Processors ← DuplicateTask(BestChromosome)          ▷ See Algorithm 11
5:     for each processor in Processors do
6:         for each task in processor.queue do
7:             TaskList.add(task)
8:         end for
9:     end for
10:    Sort tasks in TaskList according to their startTime
11:    return TaskList                                      ▷ For Distribution
```

every task in the final chromosome we check if we can duplicate it's dependencies, then we check if we can move or duplicate the task on another processor. Before duplicating any task or dependency, we check if the processor assigned to the task is not assigned to the processor our algorithm is checking. If yes, then we skip the current processor from the list and move on to the next one. It saves us time checking for a free slot on the same processor as we want to make sure that a task doesn't execute on the same processor more than once. We check for free slots[2] on the processors. For duplicating a task or dependency, we check for certain conditions (Algorithm 12), like if the estimate [3] completion time of the task is less than or equal to the available slot size, and if the free slot's starting point is greater than or equal to it's dependencies finish times. If this is true, then we duplicate the task or dependency task to the current slot. Algorithm 13 finds the minimum completion/end time for a task in the whole schedule. The algorithm determines the tasks minimum end time.

[2]A free slot is the idle time of processor's schedule length between two tasks.

[3]which can be obtained by dividing the task length by the CPU performance factor.

Algorithm 11 Duplicate Task

```
procedure DuplicateTask(BestChromosome)
    Tasks ← BestChromsome.Tasks
    Processors ← BestChromsome.Processors
    for each Task in Tasks do
        for each Dependency in Task.dependencies do
            duplicated ← FALSE and k ← 0
            while (!duplicated) AND k < processors.size() do
                processor ← processors.get(k)
                if processor.id = Dependency.assignedProcessorId then
                    k++
                    CONTINUE        ▷ Skips the processor if task is already assigned to it.
                end if
                estExecTime = (Dependency.length/processor.performance)
                for each freeSlot in processor.freeSlots do
                    if IsEligible(processor,dependency,freeSlot) then   ▷ See Algorithm 12
                        Duplicate Dependency on freeSlot on freeSlot
                        duplicated ← TRUE    ▷ This step will affect the processor's queue
not Tasks List
                    end if
                end for
            end while
            duplicated ← FALSE and k ← 0
            while (!duplicated) AND k < (processors.size()) do
                processor ← processors.get(k)
                if processor.id = Task.assignedProcessorId then
                    k++
                    CONTINUE        ▷ Skips the processor if task is already assigned to it.
                end if
                estExecTime = (Task.length/processor.performance)
                for each freeSlot in processor.freeSlots do
                    if IsEligible(processor,Task,freeSlot) then             ▷ See Algorithm 12
                        Duplicate Task on freeSlot on processor
                        duplicated ← TRUE
                    end if
                end for
            end while
        end for
    end for
    return Processors
```

Algorithm 12 Algorithm to check eligibility of Task for duplication

```
1: procedure IsEligible($Processor$, $Task$,$freeSlot$)
2:     if Task can fit in freeSlot AND $MaximumFinishTime$ for Task.dependencies is
       $\leq$ freeSlot.Start AND Task.endTime $\geq$ freeSlot.start + estExecTime +processor.COH
       AND Task.execTime $\geq$ estExecTime +processor.COH then
3:         return TRUE
4:     end if
5:     return FALSE
```

Algorithm 13 Find Task with Minimum End Time

```
1: procedure FindMinimumEndTimeTask($Processors$, $TaskId$)
2:     int min=0
3:     Task value=null
4:     for each processor in Processors do
5:         for each taskInQueue in processor.queue do
6:             if taskInQueue.id = TaskId then
7:                 if min = 0 AND taskInQueue.endTime ¿ min then
8:                     min=taskInQueue.endTime
9:                     value=taskInQueue
10:                else if min > 0 AND taskInQueue.endTime ¡ min then
11:                    min=taskInQueue.endTime
12:                    value=taskInQueue
13:                end if
14:            end if
15:        end for
16:    end for return value
```

Chapter 7

Evaluation

I used a master-slave layout of SIPS framework (see Chapter 3) to conduct my experiments. The computing machines were naturally organized as distributed network, in which the computing machines have CPUs with different speeds and/or architectures. Each node also acts as master. However, only one master node schedules and distributes the tasks. SIPS created a task graph using abstract syntax tree. This graph initially resides on a designated master processor or CPU. The master processor then splits the tasks and allocates these tasks to slave processors, CPUs in the network. Slave processors execute the tasks and allocate resources according to the nature of the task.

As mentioned there are three parts to my solution to provide better scheduling. The first part is to collect information about the nodes and resources available in the system. This was done using agents, explained in Chapter 4. The second part of the solution finds a better schedule using a genetic algorithm (GA) while considering performance, the architecture of the node, communication latency between nodes, nature of parallelism within each task and inter-dependency between tasks, as explained in Chapter 5. The third part duplicates tasks to minimize communication overhead, execution time and maximize hardware utilization, explained in Chapter 6.

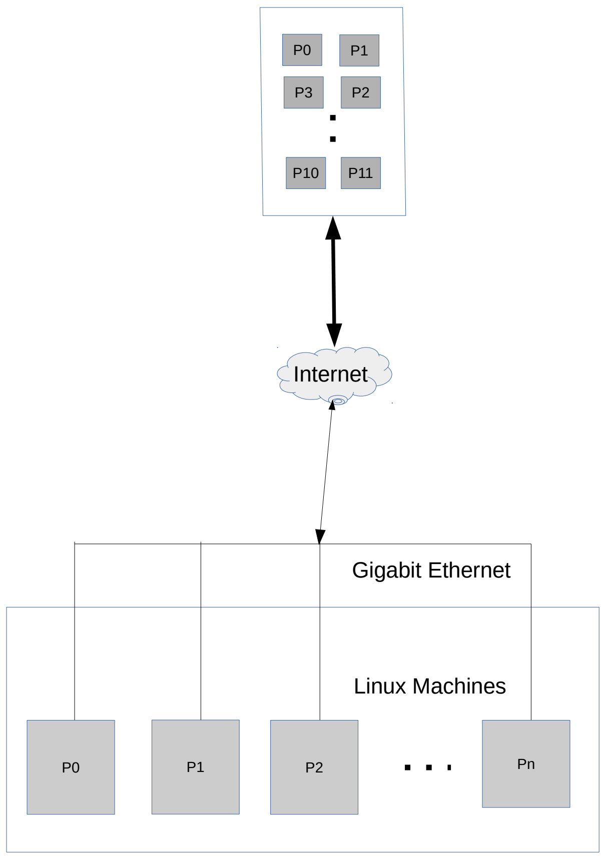

Figure 7.1: Experimental Setup

7.1 Infrastructure and Setup

For my experiments, I utilize the resources available in the Department of Computer Science. I use machines available in the Linux lab. To simulate the resource exploration, I use a remote machine (setup at my home) with 12 core processor. To serve the purpose of the master node, I use the node named “eagle” from Linux lab.

7.2 Implementation

I used Java and SIPS(Serial Algorithms In Parallel System) framework for my implementation (Chapter 3). Being the core developer of SIPS gave me the advantage of tweaking its crucial parts to gain more performance.

7.3 Evaluation

I conducted my experiments to schedule tasks in a heterogeneous[1] system consisting of multiple CPUs of different architectures (number of cores, number of threads, speed, cache size etc). I evaluate the proposed scheduling technique using evaluating parameters discussed below (as used in [13]).

1. **Total Execution Time of Tasks** is the time taken by all the tasks in one scheduling to finish execution. It depends on the performance of processing elements.

2. **Scheduling Overhead (SOH)** is the time taken by the system in making decisions to pair tasks with particular processing elements

3. **Distribution Overhead (DOH)** is time taken to distribute the tasks to the processing elements.

4. **Disk Cache Hit/Miss Ratio** used to evaluate the re-usability of data on each node. If nodes duplicate tasks on their local system, then fetching data to fulfill dependencies

[1]System is heterogeneous because nodes available in Linux lab had different CPU than Remote server, other level of heterogeneity is introduced with varying performances of all the nodes.

becomes unnecessary and the local cache can be exploited to minimize communication overhead and execution time.

5. **Average CPU usage** represents average system load in the last 1 minute, where 0.0 means very low or no system load and 1.0 or any number greater than 1.0 means high system load. Higher number also means CPU is getting flooded with requests and waiting time for processes is getting higher as CPU is being used by other processes or subtasks. For each running subtask I keep track of this parameter and at the end of the execution I calculate the average load on the during execution. Then for each task I calculate the average system load collected for subtasks.

6. **Average Sleep/Idle Time** is the total time a task went into sleep during execution, to fulfill the data dependencies.

For evaluation, I compare my technique on the basis of the parameters defined above to different schedulers: [1] Chunk, Factoring, Guided Self Scheduler (GSS), Trapezoid Self Scheduler (TSS), Quadratic Self Scheduler (QSS), and Genetic Algorithm(GA) for a set of different processors for experiment 1 described in Chapter 8. For experiment 2, I only compared Chunk, GA and my proposed technique Genetic Algorithm with Task Duplication (GATDS), since the Chunk scheduling was performing better than the other standard schedulers.

Chapter 8

Experiments and Results

I conducted two experiments to analyze the advantages of the proposed scheduling technique. In the first experiment, I used SIPS's *parallel for* loop to parallelize the multiplication of two 1000x1000 matrices. This is a more data-parallel problem. I compared my proposed scheduler with Chunk, Factoring , Guided Self Scheduling (GSS), Trapezoid self scheduling (TSS), Quadratic Self Scheduling (QSS), and Genetic Algorithm (GA) alone. To test the technique for problems with dependencies, in the second experiment, I considered merge-sort. I used SIPS task to parallelize the sorting of 20 arrays of different sizes. The algorithm for the second experiment performs sorting in multiple stages, by combining the results of previous stage, which simulates the dependency problem with 39 tasks. As I am using shared resources, I ran several iterations and averaged the results collected in the experiments.

8.1 Experiment 1 (Matrix Multiplication)

In this experiment, I exploit parallelism in the matrix multiplication problem. I use two 1000*1000 matrices in this experiment. I observed the performance of different schedulers in

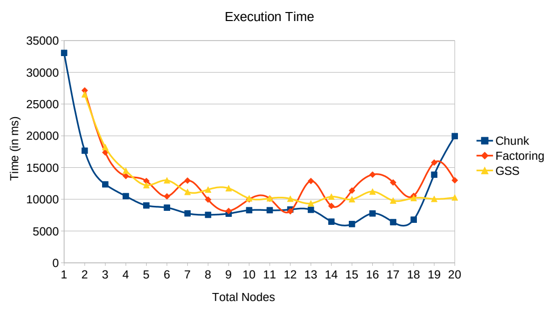

Figure 8.1: Experiment 1 - Execution time of Chunk, Factoring and GSS

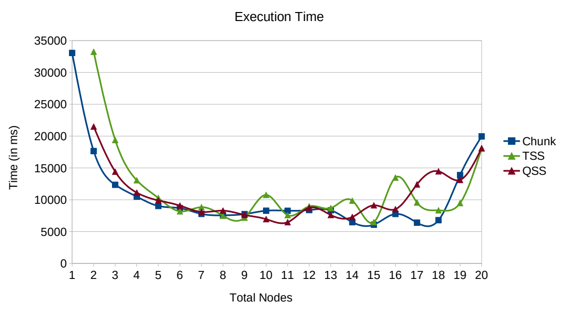

Figure 8.2: Experiment 1 - Execution time of Chunk, TSS and QSS

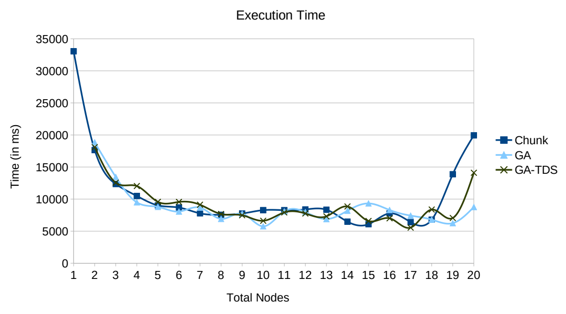

Figure 8.3: Experiment 1 - Execution time of Chunk, GA and GATDS

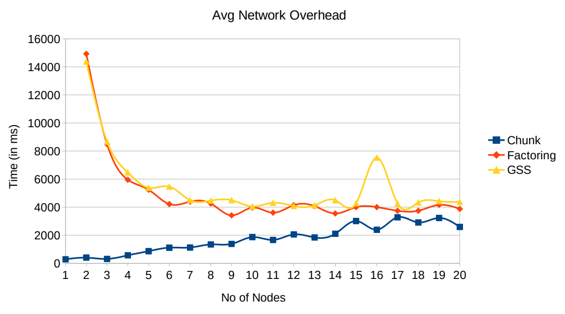

Figure 8.4: Experiment 1 - Average Network Overhead of Chunk, Factoring and GSS

terms of execution time, communication overhead (also referred as network overhead), disk cache hit-miss ratio, average sleep/idle time , average upload speed, and average CPU load. To present the data in a clean way, I divided the graph of each result into three groups. The three groups are Chunk, Factoring, GSS and Chunk, TSS, QSS and Chunk, GA , GATDS. As can be seen Chunk is common in all. The reason behind is that results produced by Kumar et al [1] reflect that Chunk performs best, which I also found true in my initial experiments. So my target is to check the performance of GA and GATDS against the Chunk scheduler. But in this experiment I have also included the data I collected for other schedulers and their performance comparison against Chunk for completeness.

Figures 8.1, 8.2 and 8.3 show execution time of the sample problem on different schedulers. From Figure 8.1 it can be noticed that Chunk outperforms Factoring and GSS scheduler. In figure 8.2 TSS and QSS does beat Chunk in some cases, but Chunk performs better in 11 out of 20 cases, with the only major drop in the performance is when I schedule the problem for 19 and 20 nodes. But that drop of performance can also be observed for TSS and QSS with respect to their overall performance. Chunk and QSS perform consistently which TSS does not. The reason behind this unexpected behaviour I believe originates from the shared network and systems. But, a separate study was conducted via running this experiment using a completely dedicated system. It was found that overall TSS beats the performance of QSS in 45% of the cases and improves when more nodes are added. From these results it can be concluded that TSS does perform better than QSS.

For the GA scheduler in this experiment, it was necessary to provide a weight (length) to tasks. While all the other schedulers divide problems into smaller tasks, GA has no predefined method of doing so. To tackle this issue, I have used TSS to split the problem into smaller tasks and then schedule them using the techniques proposed in this thesis. Also as explained in Chapter 6, GATDS depends on the output of GA for the best schedule length, so GATDS schedules the tasks divided by TSS via GA. By comparing the results, GA overall beats or matches the performance of Chunk, but for the higher number of nodes (i.e. 13 and above) GATDS does better or stays marginally behind GA in the case of nodes 18, 19 and 20.

Figures 8.4, 8.5 and 8.6 show the average network/communication overhead, which is time represented in milliseconds spent sending or receiving data. Figures 8.7, 8.8 and 8.9 present the average CPU load during execution of tasks on the given number of nodes. CPU load directly reflects the over utilization or under utilization of the CPU resources. It was

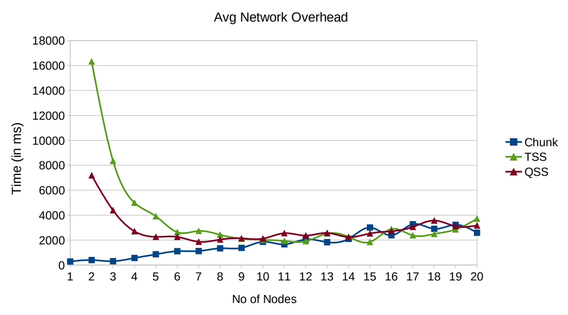

Figure 8.5: Experiment 1 - Average Network Overhead of Chunk, TSS and QSS

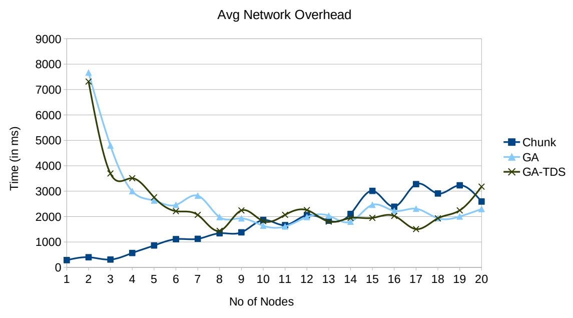

Figure 8.6: Experiment 1 - Average Network Overhead of Chunk, GA and GATDS

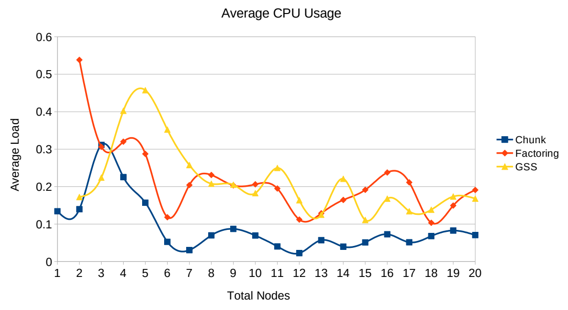

Figure 8.7: Experiment 1 - Average CPU load of Chunk, Factoring and GSS

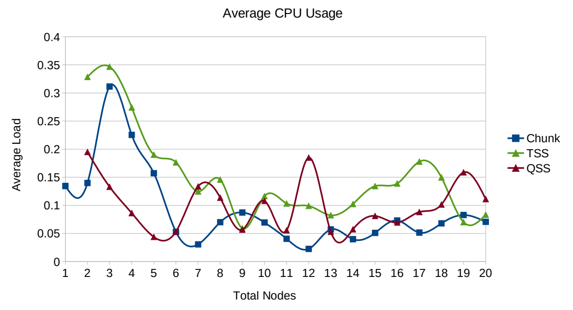

Figure 8.8: Experiment 1 - Average CPU load of Chunk, TSS and QSS

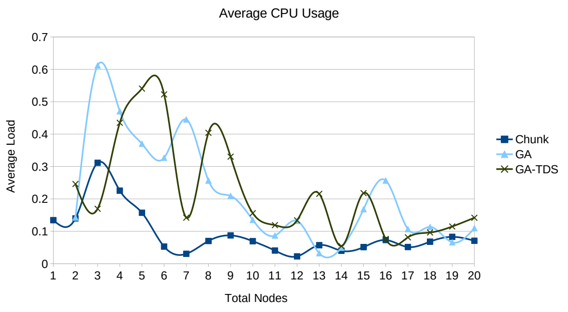

Figure 8.9: Experiment 1 - Average CPU load of Chunk, GA and GATDS

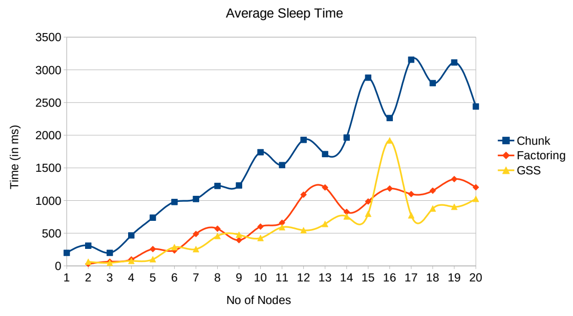

Figure 8.10: Experiment 1 - Average Sleep/idle time of Chunk, Factoring and GSS

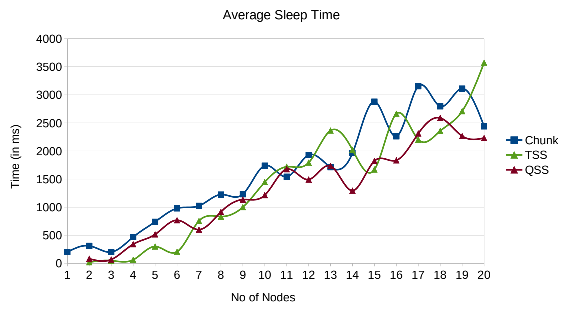

Figure 8.11: Experiment 1 - Average Sleep/idle time of Chunk, TSS and QSS

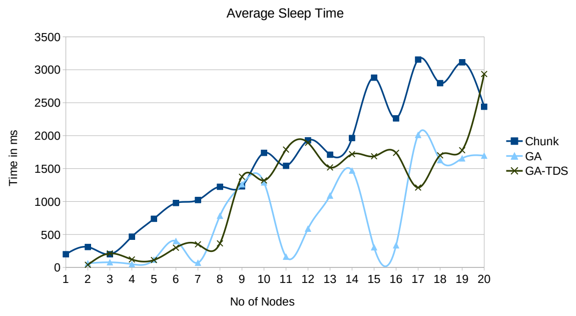

Figure 8.12: Experiment 1 - Average Sleep/idle time of Chunk, GA and GATDS

observed that CPU load crossed 0.6 while computer's responsiveness dropped. So for the experiments here any value more than 0.6 is over-utilization/flooding the CPU and lower values means under-utilization.

Average sleep/idle time is presented in Figures 8.10, 8.11 and 8.12. It can be defined as the sum of time taken when task is waiting for file request to be fulfilled and/or waiting in queue before execution. By comparing the related figures we can see that Factoring and GSS sleep less than other schedulers. My proposed scheduler does perform better than Chunk which I use as the de-facto comparison standard. GA does perform better than GATDS as per this parameter, because distribution of number of tasks per node is better for GATDS than GA.

Cache Hit-Miss ratio is represented in Figures 8.13, 8.14, and 8.15. All the schedulers perform better than the Chunk scheduler in exploiting cache, because number of tasks allocated to per node is higher for all other schedulers than Chunk scheduler. GSS and Factoring does a great job of reusing the cache, as the number of tasks is directly proportional to the number of nodes in the system, which can also be observed from Figure 8.19. If the number of chunks distributed by every scheduler is compared with their corresponding cache hit-

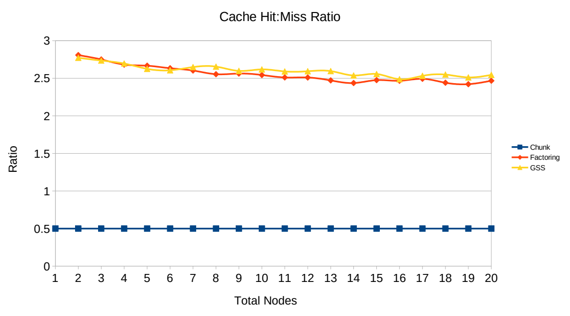

Figure 8.13: Experiment 1 - Average Cache Hit-Miss Ratio of Chunk, Factoring and GSS

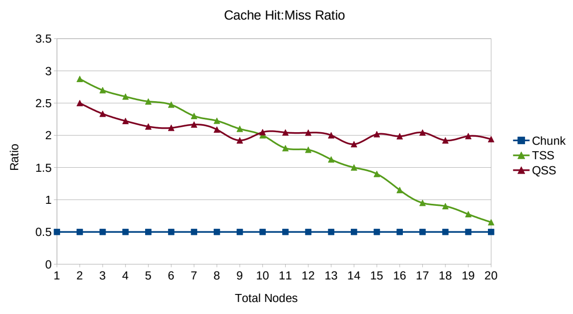

Figure 8.14: Experiment 1 - Average Cache Hit-Miss Ratio of Chunk, TSS and QSS

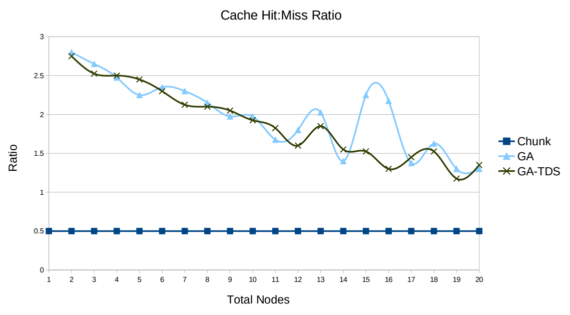

Figure 8.15: Experiment 1 - Average Cache Hit-Miss Ratio of Chunk, GA and GATDS

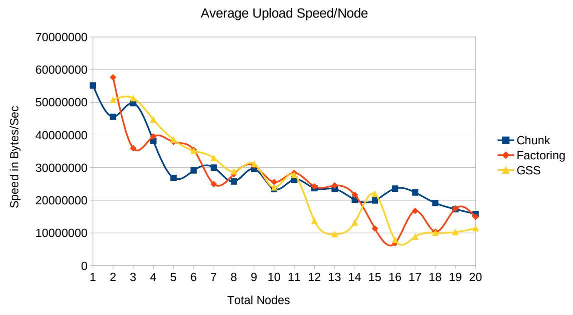

Figure 8.16: Experiment 1 - Average Upload Speed with Chunk, Factoring and GSS

miss ratio, GA and GATDS does a good job of exploiting cache, similar to TSS. The reason behind their similar performance is that they all use same kind of task sizes. The blips for GA with nodes 13, 15 and 16 is because GA allocated more tasks to same nodes.

Figure 8.19 represents the number of chunks/tasks distributed with respect to number of nodes. And Figure 8.20 presents the number of duplicate chunks created by GATDS. As the sample problem used in this experiment is data parallel and there are no inter-dependencies between tasks/chunks GATDS only creates duplicates where tasks of a schedule length on the same processor have gaps between them. So in this case, the gaps are due to high communication latency between nodes. That is another reason for the relatively poor

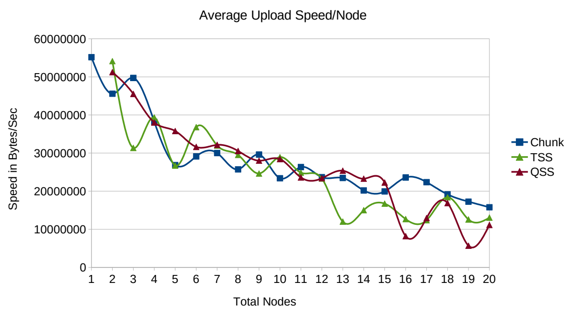

Figure 8.17: Experiment 1 - Average Upload Speed with Chunk, TSS and QSS

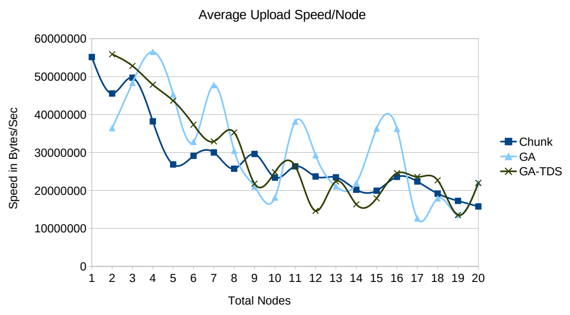

Figure 8.18: Experiment 1 - Average Upload Speed with Chunk, GA and GATDS

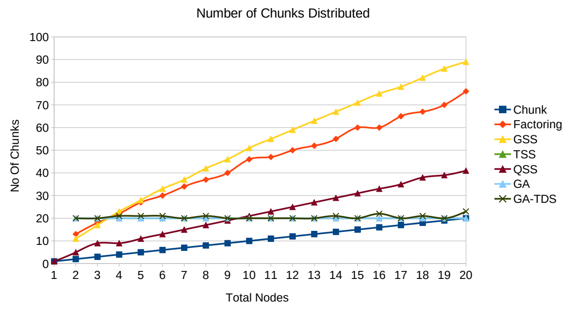

Figure 8.19: Experiment 1 - Number of Distributed Chunks/Tasks

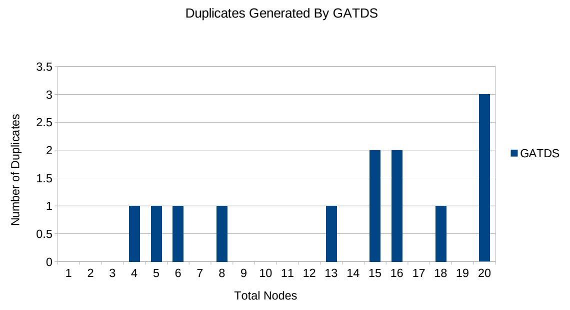

Figure 8.20: Experiment 1 - Number of Duplicates Chunks/tasks created by GATDS

performance of GATDS in terms of execution time in some cases. Looking at Figures 8.20 and 8.3 it seems that GATDS couldn't reap the benefits of duplication in Matrix Multiplication problem as expected. Thus, overall GATDS gave the same performance as GA (as with duplication part both are the same), and gave better performance than Chunk in terms of execution time, better CPU utilization and cache exploitation.

Scheduling Overhead (SOH) for Factoring and GSS is less than Chunk scheduler in most of the cases, when number of nodes were 9 or more, as shown in Figures 8.21,8.22 and 8.23. As the number of nodes increase, scheduling overhead for the Chunk Scheduler increases, whereas almost consistent for Factoring and GSS (Figure 8.21). It remains the same for

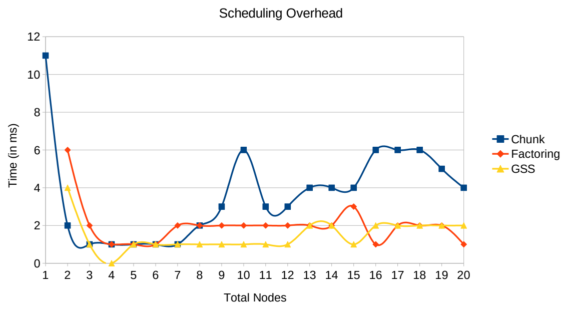

Figure 8.21: Experiment 1 - Scheduling Overhead with Chunk. Factoring and GSS

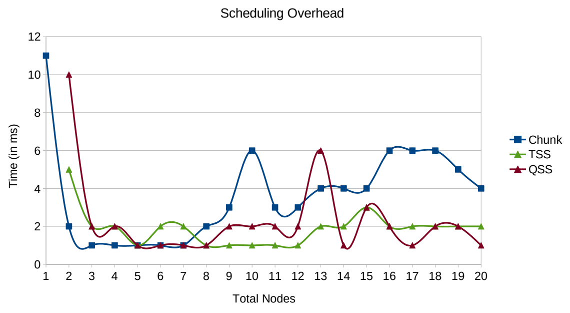

Figure 8.22: Experiment 1 - Scheduling Overhead with Chunk, TSS and QSS

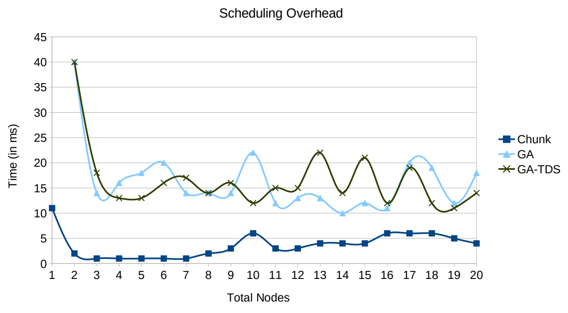

Figure 8.23: Experiment 1 - Scheduling Overhead with Chunk, GA and GATDS

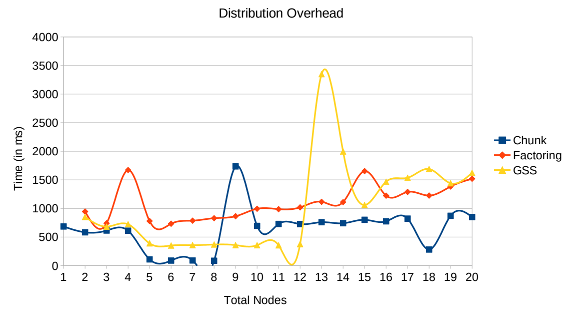

Figure 8.24: Experiment 1 - Distribution Overhead with Chunk, Factoring and GSS

TSS and QSS when compared to SOH with the Chunk scheduler. TSS and QSS take less time than chunk and perform consistently, with exceptions only for QSS when the number of nodes were 2 and 13 (see Figure 8.22). Inconsistencies for Chunk scheduler in terms of SOH were due to varying load on the master node. As we can see from Figure 8.23, GA and GATDS do not perform as well in comparison to the Chunk scheduler in terms of Scheduling Overhead, as they execute multiple stages in multiple iterations to pair tasks with the best possible nodes in such a way to keep schedule length at minimum.

Distribution Overhead (DOH) for Factoring and GSS is higher than the Chunk Scheduler as they distribute more tasks than Chunk scheduler (From Figure 8.24 and 8.19). In Figure

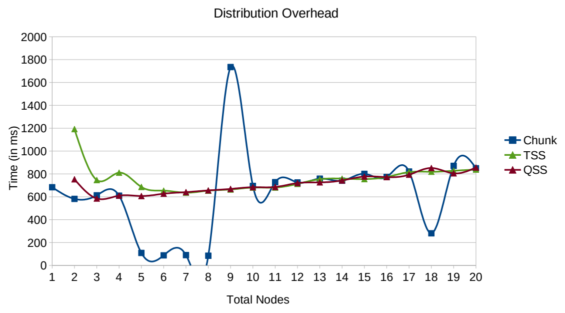

Figure 8.25: Experiment 1 - Distribution Overhead with Chunk, TSS and QSS

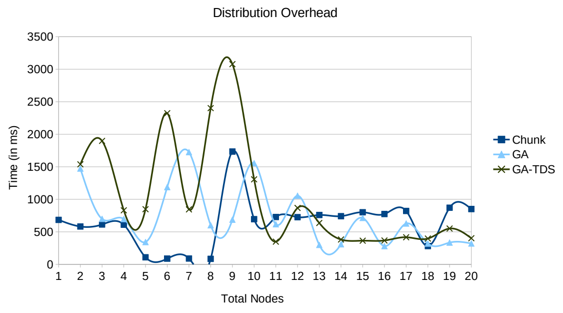

Figure 8.26: Experiment 1 - Distribution Overhead with Chunk, GA and GATDS

8.25 for TSS and QSS, DOH increases, as the number of tasks increases with the number of nodes (see Figure 8.19). DOH for GA and GATDS fluctuated for nodes 2 to 10, [1] and GA and GATDS did perform better than Chunk for the rest of the experiments. The reason behind this change in values could be the change in network load during the experiments.

8.2 Experiment 2 (Merge Sort)

In the second experiment, I used Merge Sort as the sample problem. In this problem I use 20 randomly generated arrays, sort them, merge the sorted arrays in pairs then again sort the merged arrays. This is repeated until one big sorted array is obtained. I used SIPSTask to divide this problem into sub-problems, which resulted in 39 (sub)tasks (called SIPSTasks). I assigned a weight (length) to each task as per the size of the arrays by using the API of SIPS framework. I used Chunk and GA schedulers to compare results with the my GA-TDS technique, as these two gave the best performance results in the previous experiment.

Figure 8.27 shows the execution time of the Job when I use the different schedulers. As

[1]As per my observation, The reason behind this is the shared or busy network resources. Even after multiple iterations it was difficult to correct this issue. To keep the error margin at minimum the average distribution overhead of multiple iterations was observed.

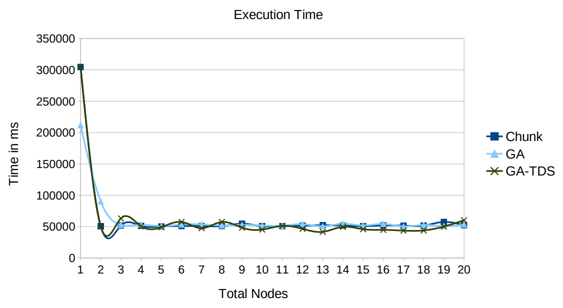

Figure 8.27: Experiment 2 - Execution Time with Chunk, GA and GATDS

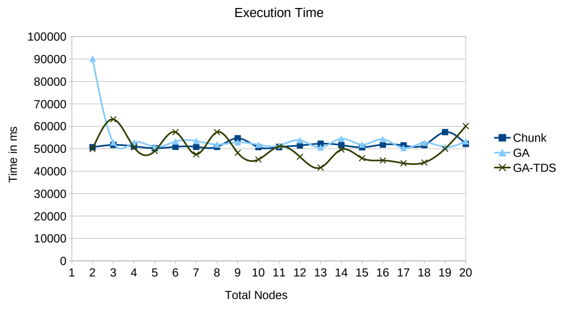

Figure 8.28: Experiment 2 - Execution Time with Chunk, GA and GATDS (without node 1)

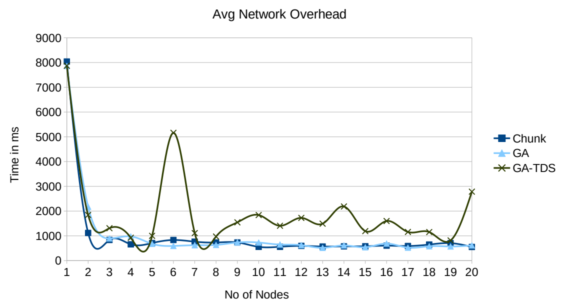

Figure 8.29: Experiment 2 - Average Network Overhead with Chunk, GA and GATDS

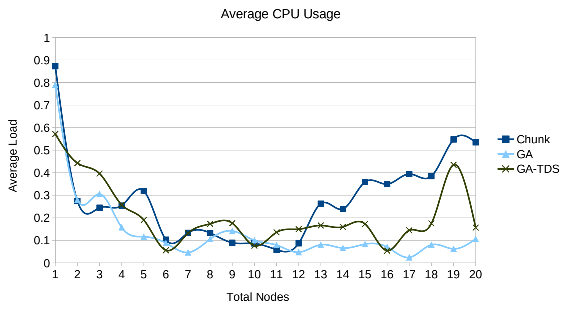

Figure 8.30: Experiment 2 - Average CPU usage/load Chunk, GA and GATDS

can be seen, it is difficult to interpret the data from Figure 8.27 clearly, because of the high difference between mean value and maximum value in the graph. So I have presented the data again after eliminating the value for node 1 in Figure 8.28, which represents the data more clearly. As can be seen from Figure 8.28, my GA-TDS technique beats or matches the performance of Chunk and GA in more than 75 % of the cases. GA and chunk gave virtually identical performance in most of the cases.

Figure 8.29 shows the average time spent on communication over the network. The reason behind the higher values for GATDS is because of the higher number of duplicate tasks (see

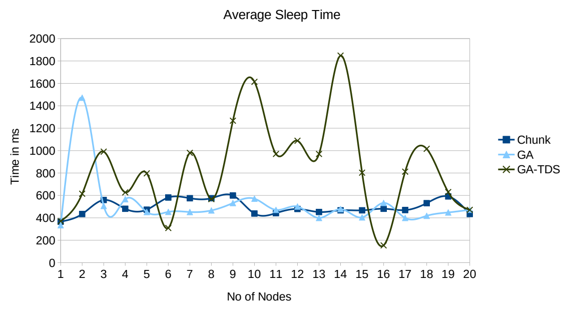

Figure 8.31: Experiment 2 - Average Sleep Time Chunk, GA and GATDS

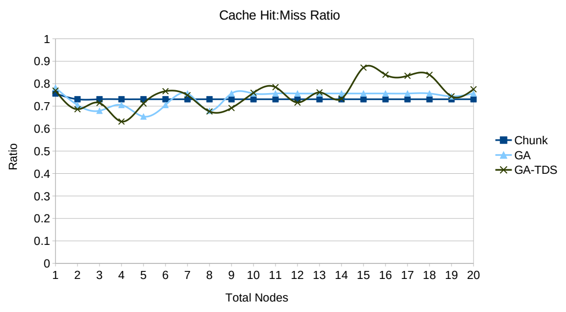

Figure 8.32: Experiment 2 - Average Cache Hit-Miss ratio with Chunk, GA and GATDS

Figure 8.35). Because of the higher number of tasks running in parallel and requesting for file/data transfer the average communication overhead is high.

Figure 8.30, represents the average CPU usage. From this figure and the reduction in total execution time, it can be deduced that GATDS does a better job at resource utilization than the other two. The reason why the results for GATDS lies between GA and the Chunk scheduler is that, Chunk scheduler tries to divide tasks equally and big tasks consume all the available resources without letting other tasks use the resources resulting in high CPU load. While GA tries to match the best resources for the task execution, because as it allocates more capable resources for execution it could result in low resource utilization. GATDS duplicates the same tasks and tries to take advantage of hardware level parallelization (multicore and multithreading), thus resulting in adequate level of hardware utilization with minimal execution time.

Figure 8.31 shows the average sleep time for tasks. GATDS tasks wait more than GA and Chunk tasks, as the number of tasks is almost double of that other scheduler's tasks (From Figure 8.35 and number of predefined tasks) in some cases. It can be observed from Figure 8.32, that GATDS does a better job of exploit caching than Chunk and GA, as the

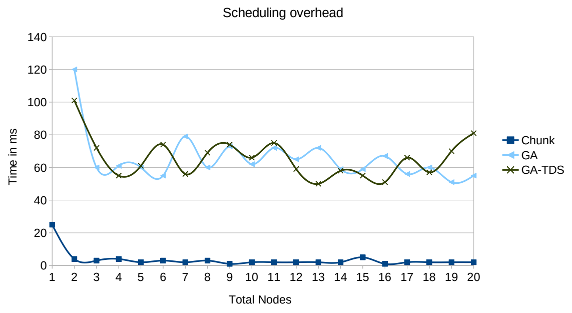

Figure 8.33: Experiment 2 - Scheduling Overhead with Chunk, GA and GATDS

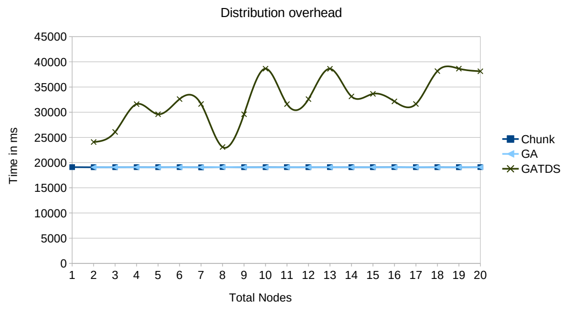

Figure 8.34: Experiment 2 - Distribution Overhead with Chunk, GA and GATDS

number of tasks distributed by GATDS is higher and the results of other tasks were locally available to successor tasks.

Figure 8.33 shows the scheduling overhead of all the schedulers. The scheduling overhead for Chunk is very low as it just pairs the tasks with the best possible nodes evenly. Whereas the values for GA and GATDS vary and are higher than Chunk. This is because GA performs multiple iterations to find the best schedule length and GATDS (additionally) follows a brute force approach to find slots in which duplicates tasks. In Figure 8.34, the distribution overhead for Chunk and GA remains same as number of tasks were constant, whereas GATDS distributes some extra tasks (because of duplication), thus takes extra times to send those tasks and related data to the assigned nodes. I observed network latency also plays a very huge role here.

Figure 8.35, presents the number of duplicate tasks generated by GATDS. GATDS as mentioned in Chapter 6, duplicates the tasks in the free slots available between tasks on processors. From the graph we can observe that with the increase in the number of nodes, the number of duplicates also increases, which i believe is because of the communication latency between nodes give rise to the number of free slots between tasks on processors.

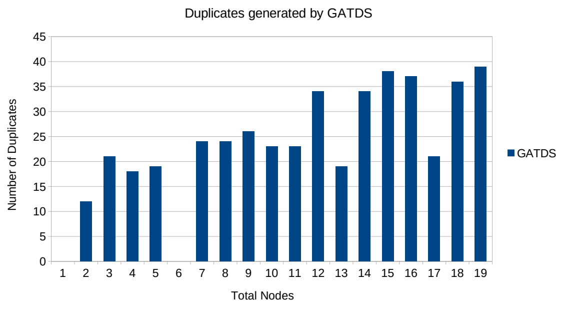

Figure 8.35: Experiment 2 - Number of Duplicates created by GATDS

8.3 Summary

In summary, the experiments indicate that for data parallel problems such as Experiment 1, GATDS performance is comparable to genetic algorithm. In these problems there is no task dependency and the communication latency is not a hindrance. Task duplication causes negative performance in these problems for higher numbers of processors. However, for problems with inter-dependent tasks such as Experiment 2, GATDS improves the performance and is better than GA. Task duplication allows processors to execute other tasks instead of idling and reduces communication latency between the processors. The execution time is thereby reduced.

Chapter 9

Conclusion

In conclusion, this thesis proposed a scheduling strategy that uses agents to collect information about the network, a genetic algorithm to determine a good schedule to map the tasks on to the network machines and use of the task duplication scheme to further improve the scheduling to increase reliability and minimize communication between processors. The SIPS (Serial Algorithms In Parallel System) framework was used to exploit parallelism using abstract syntax trees generated directly from the source code. The proposed scheduling strategy, Genetic Algorithm with Task Duplication Startergy (GATDS) was tested on two problems, one task-independent and other task-dependent. GATDS performs better for task-dependent problems such as merge-sort compared to a pure genetic algorithm.

Future Work: A separate study could be conducted to further study the overheads mentioned in this thesis. I also believe that GATDS can be improved further by exploiting parallelization within the technique or finding a better way to duplicate tasks. As mentioned in Chapter 2, parallel scheduling algorithms already exist, thus it will also be interesting to see if we can minimize scheduling overhead for GA and GATDS by parallelizing their execution.

Currently, SIPS does not support GPU execution. One other area of future research would be to widen the usability of the SIPS framework to accelerators. I also believe that my results were affected by the shared usage of the available resources. By conducting the experiments in controlled and dedicated environment, i expect to generate different results than the ones i have presented in this thesis.

Appendix A

Source Code

All the code we have used in our experiments are hosted on Github. All the source code is available under GNU GPL License v3. Below are the links to the corresponding modules.

SIPS-samples: https://github.com/deepsidhu1313/SIPS-samples

SIPS-lib: https://github.com/deepsidhu1313/SIPS-lib

SIPS-Schedulers: https://github.com/deepsidhu1313/SIPS-Schedulers

SIPS-Node: https://github.com/deepsidhu1313/SIPS-Node

SIPS-Run: https://github.com/deepsidhu1313/SIPS-Run

A.1 Sample Manifest File

```
// sample manifest.json
    {
        "PROJECT": "MatMul",
        "ARGS": [],
        "LIB": ["SIPS-lib-0.2-SNAPSHOT-jar-with-dependencies.jar"],
        "OUTPUTFREQUENCY": 100,
        "JVMARGS": [],
        "SCHEDULER": {
            "MaxNodes": "4",
            "Name": "in.co.s13.sips.schedulers.Chunk"
        },
        "MASTER": {
            "HOST": "127.0.0.1",
            "API-PORT": "13139",
            "JOB-PORT": "13136",
            "API-KEY": "ff21930e-2f22-4d57-8ca8-0e1b0d4b4e31",
        },
        "MAIN": "MatMul",
        "ATTCH": []
    }
```

Bibliography

[1] A. Kumar, H. Singh, and N. Singh, "Sips: A framework to run serial algorithms in parallel systems using abstract syntax tree," *International Journal of Applied Engineering Research*, vol. 13, no. 8, pp. 6165–6176, 2018.

[2] N. Singh, "Framework for implementing serial algorithms in parallel system using abstract syntax tree," Master's thesis, Department of Computer Science,Guru Nanak Dev University, Amritsar, Punjab, India, 2014.

[3] M. Daga, A. M. Aji, and W. chun Feng, "On the efficacy of a fused CPU+GPU processor (or apu) for parallel computing," in *Symp on Appl. Accelerators in HPC*, July 2011.

[4] K. Nilakant and E. Yoneki, "On the efficacy of apus for heterogeneous graph computation," in *Fourth Workshop on Systems for Future Multicore Architectures*, Amsterdam, Netherlands, 2014.

[5] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, 2017, pp. 5–14.

[6] A. Y. Zomaya, C. Ward, and B. Macey, "Genetic scheduling for parallel processor

systems: comparative studies and performance issues," *IEEE Transactions on Parallel and Distributed systems*, vol. 10, no. 8, pp. 795–812, 1999.

[7] S. Ranaweera and D. P. Agrawal, "A task duplication based scheduling algorithm for heterogeneous systems," in *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS).* Cancun, Mexico: IEEE, May 2000, pp. 445–450.

[8] D. J. Lilja, "Exploiting the parallelism available in loops," *Computer*, vol. 27, no. 2, pp. 13–26, 1994.

[9] J. Diaz, S. Reyes, A. Nino, and C. Munoz-Caro, "A quadratic self-scheduling algorithm for heterogeneous distributed computing systems," in *Cluster Computing, 2006 IEEE International Conference on.* IEEE, 2006, pp. 1–8.

[10] M. Aggarwal, R. D. Kent, and A. Ngom, "Genetic algorithm based scheduler for computational grids," in *19th International Symposium on High Performance Computing Systems and Applications.* IEEE, May 2005, pp. 209–215.

[11] J. Sun, X. Dong, X. Zhang, and Y. Wang, "An availability approached task scheduling algorithm in heterogeneous fault-tolerant system," in *9th IEEE International Conference on Networking, Architecture, and Storage (NAS).* Tianjin, China: IEEE, August 2014, pp. 275–280.

[12] K. J. Naik and N. Satyanarayana, "A novel fault-tolerant task scheduling algorithm for computational grids," in *15th International Conference on Advanced Computing Technologies.* Newboyanapalli, Rajampet, India: IEEE, September 2013, pp. 1–6.

[13] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506–521, 1996.

[14] ——, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, pp. 381–422, 1999.

[15] I. Ahmad, Y.-K. Kwok, and M.-Y. Wu, "Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors," in *Proceedings of the Second International Symposium on Parallel Architectures, Algorithms, and Networks.* Beijing, China: IEEE, June 1996, pp. 207–213.

[16] H. Oh and S. Ha, "A static scheduling heuristic for heterogeneous processors," in *European Conference on Parallel Processing.* Lyon, France: Springer, August 1996, pp. 573–577.

[17] J. V. Lima, T. Gautier, V. Danjean, B. Raffin, and N. Maillard, "Design and analysis of scheduling strategies for multi-cpu and multi-gpu architectures," *Parallel Computing*, vol. 44, pp. 37–52, 2015.

[18] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[19] J. V. F. Lima, T. Gautier, N. Maillard, and V. Danjean, "Exploiting concurrent gpu operations for efficient work stealing on multi-gpus," in *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, Oct 2012, pp. 75–82.

[20] O. Beaumont, L. Eyraud-Dubois, A. Guermouche, and T. Lambert, “Comparison of static and dynamic resource allocation strategies for matrix multiplication,” in *26th IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).* Paris, France: IEEE, May 2015.

[21] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke, “Impact of workload and system parameters on next generation cluster scheduling mechanisms,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 9, pp. 967–985, 2001.

[22] I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and W. Shu, “Automatic parallelization and scheduling of programs on multiprocessors using casch,” in *Proceedings of the 1997 International Conference on Parallel Processing.* Yaroslavl, Russia: IEEE, September 1997, pp. 288–291.

[23] A. Sohn and H. Simon, “Jove: A dynamic load balancing framework for adaptive computations on an sp-2 distributed memory multiprocessor,” in *Technical Report*, New Jersey, USA, 1994.

[24] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, “Starpu: a unified platform for task scheduling on heterogeneous multicore architectures,” *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[25] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, “Data-aware task scheduling on multi-accelerator based platforms,” in *IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS).* Shanghai, China: IEEE, December 2010, pp. 291–298.

[26] L. Cohen. (2018) Jppf. [Online]. Available: http://jppf.org

[27] M. K. Dhodhi and I. Ahmad, "A multiprocessor scheduling scheme using problem-space genetic algorithms," in *IEEE International Conference on Evolutionary Computation*, vol. 1. Perth, Western Australia: IEEE, December 1995, p. 214.

[28] T. Yang and A. Gerasoulis, "List scheduling with and without communication delays," *Parallel Computing*, vol. 19, no. 12, pp. 1321–1344, 1993.

[29] M. A. Palis, J.-C. Liou, and D. S. L. Wei, "Task clustering and scheduling for distributed memory parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 1, pp. 46–55, 1996.

[30] M. Bohler, F. W. Moore, and Y. Pan, "Improved multiprocessor task scheduling using genetic algorithms." in *FLAIRS Conference*, Orlando, Florida, July 1999, pp. 140–146.

[31] S. Kim and J. B. Weissman, "A genetic algorithm based approach for scheduling decomposable data grid applications," in *International Conference on Parallel Processing.* Montreal, Quebec, Canada: IEEE, August 2004, pp. 406–413.

[32] A. J. Page, T. M. Keane, and T. J. Naughton, "Multi-heuristic dynamic task allocation using genetic algorithms in a heterogeneous distributed system," *Journal of parallel and distributed computing*, vol. 70, no. 7, pp. 758–766, 2010.

[33] F. Pinel, B. Dorronsoro, and P. Bouvry, "Solving very large instances of the scheduling of independent tasks problem on the gpu," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 101–110, 2013.

[34] S. Solomon, P. Thulasiraman, and R. Thulasiram, "Collaborative multi-swarm pso for task matching using graphics processing units," in *Proceedings of the 13th annual con-*

ference on Genetic and evolutionary computation. Dublin, Ireland: ACM, July 2011, pp. 1563–1570.

[35] M. S. Sidhu, P. Thulasiraman, and R. K. Thulasiram, "A load-rebalance pso heuristic for task matching in heterogeneous computing systems," in *IEEE Symposium on Swarm Intelligence (SIS).* Singapore: IEEE, April 2013, pp. 180–187.

[36] A. A. Beegom and M. Rajasree, "A particle swarm optimization based pareto optimal task scheduling in cloud computing," in *International Conference on Swarm Intelligence.* China: Springer, October 2014, pp. 79–86.

[37] F. Song and J. Dongarra, "A scalable framework for heterogeneous gpu-based clusters," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures.* USA: ACM, June 2012, pp. 91–100.

[38] M. Dorigo and T. Stützle, "The ant colony optimization metaheuristic: Algorithms, applications, and advances," in *Handbook of metaheuristics.* Springer, 2003, pp. 250–285.

[39] Z. Jin, Z. Yang, and T. Ito, "Metaheuristic algorithms for the multistage hybrid flowshop scheduling problem," *International Journal of Production Economics*, vol. 100, no. 2, pp. 322–334, 2006.

[40] T. Liao, T. Stützle, M. A. M. de Oca, and M. Dorigo, "A unified ant colony optimization algorithm for continuous optimization," *European Journal of Operational Research*, vol. 234, no. 3, pp. 597–609, 2014.

[41] G. Singh, "Job Scheduling In Parallel Computing using Task Duplication Technique,"

Master's thesis, Department of Computer Science,Guru Nanak Dev University, Amritsar, Punjab, India, 2015.

[42] J. Bruno, E. G. Coffman Jr, and R. Sethi, "Scheduling independent tasks to reduce mean finishing time," *Communications of the ACM*, vol. 17, no. 7, pp. 382–387, 1974.

[43] J. Błazewicz, J. Weglarz, and M. Drabowski, "Scheduling independent 2-processor tasks to minimize schedule length," *Information Processing Letters*, vol. 18, no. 5, pp. 267–273, 1984.

[44] R. L. Sri and N. Balaji, "Meta-heuristic hybrid dynamic task scheduling in heterogeneous computing environment," in *International Conference on Computer Communication and Informatics (ICCCI)*. Tamil Nadu, India: IEEE, January 2013, pp. 1–6.

[45] S. A. Kazarlis, A. Bakirtzis, and V. Petridis, "A genetic algorithm solution to the unit commitment problem," *IEEE transactions on power systems*, vol. 11, no. 1, pp. 83–92, 1996.