International Journal of Applied Engineering Research ISSN 0973-4562 Volume 13, Number 8 (2018) pp. 6165-6176

SIPS: A Framework to Run Serial Algorithms in Parallel Systems using Abstract Syntax Tree

Anil Kumar[1,*], Hardeep Singh[2] and Navdeep Singh[3]

[1]Assistant Professor, Department of Computer Engineering and Technology, Guru Nanak Dev University, Amritsar, India.
[2]Department of Computer Science, Guru Nanak Dev University, Amritsar, India.
[3]Department of Computer Science, University of Manitoba, Canada.

**Corresponding author*

Abstract

A new framework is proposed for the implementation of Serial Algorithms in Parallel Systems (SIPS) using Abstract Syntax Tree. Performance of computer can be improved by the use of two basic techniques: better and faster hardware, and exploiting parallelism for concurrent execution. These techniques are not mutually exclusive. Parallelism in loops is exploited to reduce overall computation time, as well as to increase speedup. Abstract Syntax Tree (AST) generated by parsing source code can provide useful information related to loops present in source code. This work targets to design such a framework which can reduce the efforts of a programmer to design special parallel applications intended to be used on parallel systems. Redesigning or Re-engineering serial applications for parallel computing increase the cost of a project as well as requires more human efforts & time to achieve the desired target. Implementation of serial algorithms in parallel fashion without making any major changes on programmer's end will surely saves a lot of human effort and time.

Keywords: High performance computing, distributed computing, scheduling, load balancing, loop-parallelization.

INTRODUCTION

Frameworks for parallel computing have recently become popular as a means for preserving parallel algorithms as reusable components. The purpose of a parallel framework is ease-of-use and code reuse. The framework can be summarised in two folds: by providing property of code reuse, and total control & monitoring of entire calculation. Framework which can exploit parallelism for concurrent execution will help to achieve high performance in parallel systems. Exploiting parallelism in loops can reduce overall computation time and a significant speedup.

Frameworks are useful in providing reusable components for further development, thus saving a lot of time at developers end. Same components can be reused to perform different tasks. In parallel systems, frameworks can be helpful in controlling and monitoring entire process executing on different nodes in the system. Each node in the system can be monitored which is thus helpful in performing tasks such as scheduling and load balancing. The user needs to be only aware of the interface of the framework and not the unnecessary details of the parallel system i.e. algorithms used and data level detail, that is exchanged between different components in the system. The frameworks are fundamentally a different approach than that of a compiler, parallelizing or otherwise. The frameworks are more responsible and participate in the creation phase, framing phase and as well as the execution phase. Compilers are not involved in what happens during the execution of the programs they compile.

Redesigning or re-engineering serial applications for parallel computing increases the cost of a project as well as requires more human efforts & time to achieve the desired goal. Software evolution becomes easy to understand when mining of software repositories is done at the source code level The main objective of this work is to design a framework which can reduce the efforts of a programmer to design special parallel applications targeted to be run on parallel systems. A framework has been proposed based on all the mentioned elements which will help developers to implement serial algorithms in parallel systems.

RELATED WORK

Many different type of frameworks have been proposed and implemented by many authors and companies in the past with a variety of objectives. Lindholm et al. [16] proposed CUDA (Compute Unified Device Architecture) for general purpose computing on GPUs to explore the unknown processing potential of underlying graphic cards to largely enhance the performance of video processing. Wang et al. [2] proposed that the dyanamic evolution of parallelism in applications like GPU is brought in by CUDA Dyanamic Parallism model.

Jolivet et al. [3] proposed the HPDDM (High Performance Unified Framework for Domain Decomposition Methods) for huge parallel computing. It is an efficient implementation of various domain decomposition methods. Tournier et al. [1] proposed HPDDM and FreeFem++ tools to facilitate the microwave tomography for brain stroke imaging.

Gonzalez et al. [4] proposed 'GraphX' which is a distributed data flow framework for graph processing. There is a convenience of specialized graph processing systems in modern general purpose data flow systems, upon which many authors have argued. On top of Apache Spark, GraphX is built which is enclosed graph processing framework and it is largely used in distributed data flow system. A common composed graph abstraction which is enough for expressing existing graph APIs is presented in 'GraphX'. This abstraction can be implemented by making use of basic data-flow

operators. Recasting of graph-specific optimizations as materialized view maintenance as well as distributed join optimizations is done by 'GraphX' for achieving performance parity. It has brought low-cost fault tolerance to graph processing by utilizing the advances of distributed data-flow frameworks. Poulson et al. [5] proposed 'Elemental' which is a new framework for performing distributed memory dense matrix computations. Parallelization of computations of dense matrix to distributed memory architectures, is one of the finest understood domains of parallel computing. ScaLAPACK and PLAPACK are the two packages that were developed in the mid-1990s and still are considered good for regular use. Along with the arrival of many-core architectures, these packages need to be revised, as the conventional MPI-based approaches will probably need an extension. Ou, Yulong, et al. [6] proposed a multiple master-slave strategy in 'LPFSC'. It is a light weight parallel framework for supercomputing. Programmers can easily plan tasks to run on supercomputer to achieve reduced overhead cost issues, like resource management, system reliability and job scheduling for a large supercomputer. This strategy is particularly important to maintain good performance under varied jobs and hardware platform features.

H. Esplanade et al. [8] proposed P2G which is a framework intended to process the distributed real-time multimedia data to meet the rising needs of the computations to be performed on processing of multimedia data. P2G framework suffers from the constraint of executing complex workloads randomly. Therefore Zhou et al. [9] came up with a new framework known as P2P (peer-to-peer). Authors proposed a simple model for chunk scheduling strategies in Peer-to-Peer streaming. The authors have illustrated a simple stochastic model which can be utilized for the comparison of different downloading strategies to random peer selection. Two factors: (1) Scalability for the propagation of the file chunks to the maximum possible peers, and (2) the strategy for getting the chunks required urgently from maximizing playback continuity are emphasized. Zhang et al. [7] proposed 'iMapReduce' which is a distributed computing framework for iterative computation. The pervasiveness of the relational data in most applications like data mining or social network analysis, which may consists around hundreds of millions of relations, has raised the demand for designing the framework for distributed computing to process these data on a large cluster. Nevertheless, parsing the relational data iteratively is required by many relational data based applications and also these applications require iterations to perform different operations on these data. 'iMapReduce', besides providing a full support for iterative processing automatically, also permits the users for specifying the iterative operations with map and reduce functions. It considerably enhances the performance of iterative algorithms by elimination of the shuffling of the static data in the shuffle stage of MapReduce, permitting asynchronous execution of each iteration, and by the reduction of the overhead of new task in every iteration. Honjo et al. [10] proposed a unified parallel framework that supports heterogeneity as well as fault tolerance in MPI programs while working on a range of parallel computing platforms. It does the scheduling of a large number of parallel as well as sequential jobs in an optimal manner. The jobs are submitted by numerous users on a heterogeneous parallel computing environment. Jose et al. [11] proposed P2PComp framework which is based on philosophy of the pure peer-to-peer model, is used for developing and executing parallel and distributed applications using the peer-to-peer computing model. It supports the functions for starting and monitoring processes, searching resources and communicating by message passing.

Ma et al. [12] presented a novel system framework for high performance 'Geo-computing'. It resolves the issues such as the problems with the provision of a high performance Geo-computing system having high speed as well as provides the ease of use for the researchers. HPGOSS (High Performance Geo-data Object Storage System) which relies on parallel file system is employed to eliminate the bottleneck of I/O performance along with that for handling the challenge of the data management that results from the close relevance between remote sensing & geo-information image data. Yu et al. [14] proposed Dryad LINQ which is a system for general-purpose distributed data-parallel computing making use of a high-level language. Basically, Dryad LINQ is a set of language extensions enabling a new programming model intended for distributed computing in large scale. It makes the generalization of the prior execution environments like MapReduce, SQL, as well as Dryad by two means: (1) implementing a significant data model of strongly typed .NET objects, and (2) by aiding general-purpose declarative along with imperative procedures and functions on datasets inside a conventional high-level programming language. A Dryad LINQ program is a sequential program comprised of LINQ expressions which perform random side-effect-free transformations on datasets, at the same time they can be written as well as debugged by making use of standard .NET development toolkits. Automatically and transparently translating the data-parallel program's portions of the program to a distributed execution plan is what a Dryad LINQ system is intended to do and then passed to the Dryad execution platform.

Hecht et al. [17] proposed FreeFem++ which focuses on parallel simulation of equations from physics by finite element method is free software including versions for console and MPI. Goodale et al. [19] proposed 'Cactus', which is a framework for developing numerous computing applications in science as well as engineering. It includes relativity, astrophysics, and also chemical engineering. Authors have been motivated to propose such a framework on realizing the need for such frameworks in supporting high performance, multi-platform applications across different communities. Then the design of the latest release of Cactus (Version 4.0) is described in which a total rewrite of earlier versions is there, enabling multi-language, highly modular, parallel applications to be developed by anyone. By making use of abstractions extensively, the authors have detailed how the latest advances in computational science are provided by them, like high performance I/O layers as well as interchangeable parallel data distribution, while hiding the majority details of the underlying computational libraries from the application developer. Sorasen et al. [22] proposed SWIPP (Switched Interconnection of Parallel Processors) which is a

computer framework providing feasibility for development of parallel systems in accordance with BSP (Bulk Synchronous Parallel) computing model. Challenging applications are expressed as directed graphs where the nodes comprise the sub-tasks; which are interdependent in nature. Foster et al. [23] proposed GLOBUS which is a meta-computing infrastructure toolkit. The authors developed GLOBUS to handle the difficulties posed by dynamic as well as the heterogeneous meta-computing environment. Globus system is designed for achieving the vertical integration of network, middleware with the application. Yau et al. [27] proposed PROOF which is a framework for developing software for distributed parallel computing systems and based on the parallel object-oriented functional computational model. The authors have used the technique which allowed program execution to take place independently of the configuration of the computing system. Also, the programmer need not be concerned about the parallelism in the application or the architectural specifications. Sunderam et al. [29] proposed PVM which is a programming environment intended for the developing as well as executing large parallel or concurrent applications consisting of many interacting, but relatively independent, components. It is designed for operating on a collection of heterogeneous computing elements which are interconnected to each other by one or more networks. The processors which are participating in PVM may be multiprocessors, scalar machines, or special-purpose computers. The participating processors enable application components for execution on the architecture which is most appropriate to the algorithm. A general and straightforward interface is provided by PVM, which allows the description of various types of algorithms and their interactions. The underlying infrastructure of PVM allows the execution of applications on a virtual computing environment which supports multiple parallel computation models. PVM comprises facilities for sequential, concurrent, or conditional execution of application components.

JPPF(Java Parallel Processing Framework) [33] refers to "Write once,Deploy once,Execute everywhere". It reduces the processing time of applications with large rfequirement of processing capacity. So that they can execute everywhere. This framework is also the comparison framework in this work.

Scheduling in a parallel system plays a major role in the performance of a parallel computing framework, as it manages the load balancing among the available nodes in parallel systems. A variety of different scheduling strategies can be used to determine which iterations should be executed by which processors.

Chiang et al. [13] proposed goal-oriented strategies for balancing multiple scheduling performance requirements and efficient management of resources on parallel computer systems, thus allowing the parallel systems to achieve high-level performance. Diaz, Javier, et al. [18] presented a general formulation of the self-scheduling problem, deriving a new, quadratic, self-scheduling algorithm. The authors presented this general formulation as scheduling algorithms perform a vital role within heterogeneous computing systems. The proposed algorithm proved to be much better when compared to the earlier algorithms when implemented by allocating sets of tasks in an internet-based grid of computers. Lilja, David J et al. [24] discussed the exploitation of parallelism in Loops can reduce the computational time by running each or set of iterations on different node in the parallel computer system. The body of a loop may be executed many times in applications to compute larger data. To exploit the maximum parallelism available in loops, knowledge of data dependence which exists between different elements of a program is necessary. The three types of dependencies are resource, data, and control. The actual parallelism available in a program is limited by its dependencies. By checking data dependence, we can figure out the maximum parallelism available in the loops. This data dependence may also consist of cross-iteration data dependence between elements. The longest dependence chain produced by different iterations in loops is called the critical path. The minimum time to execute the entire loop is the time required to execute the longest dependence chain. Shared-memory multiprocessors exploit coarse-grained parallelism by distributing entire loop iterations to different processors. Do-across scheduling can be used to distribute 'for loops' with cross-iteration dependencies to separate processors with a delay of time between each set of iteration so that dependence can be solved. Do-all loop scheduling can be used to schedule a loop with no cross-iteration dependencies, and where all iterations are logically independent and can be executed in any order on different nodes. As source of motivation, and base of this research, program transformation techniques can be used to develop compilation techniques that automatically transform programs to execute on parallel computers. Markatos et al. [25] proposed the utilization of processor affinity in loop scheduling on shared-memory multiprocessors as loops are the sole biggest source of parallelism in numerous applications. A common way for exploiting this parallelism is the execution of loop iterations in parallel on different processors. The authors have considered a new aspect to the problem of loop scheduling on shared-memory multiprocessors: communication overhead created by accesses to non-local data whereas the previous approaches to perform loop scheduling tried in achieving the minimum completion time the distribution of the workload as evenly as possible while minimizing the number of synchronization operations needed. The authors have showed that conventional algorithms for loop scheduling had always ignored the location of data when assignment of iterations to processors is done. By not taking into consideration the location of data, a considerable performance penalty is incurred on modern shared-memory multiprocessors. Therefore the authors have proposed a new loop scheduling algorithm in this paper which tries in simultaneously balancing the workload, minimizing synchronization, as well as co-locating loop iterations with the necessary data. Tzen et al. [26] proposed 'Trapezoid self-scheduling' which is a self-scheduling scheme for arbitrary parallel nested loops in shared-memory multiprocessors, usually the main source of parallelism is the loops in any parallel program. Load balancing can be achieved by dynamically allocating loop iterations to processors at the cost of run-time scheduling overhead. In the 'Trapezoid self-scheduling' approach, the best trade-off between the balanced

workload as well as scheduling overhead can be achieved by linearly reducing the chunk size at run time. This approach can be effectively implemented in any parallel compiler due to its flexibility as well as simplicity. Efficient memory management is also allowed by the small and predictable number of chores in a static fashion. Before that Hummel at al. [28] proposed a factoring method for deciding chunk size to improve load balancing. According to the authors, the next chunk size can be calculated by the formula *[chunk* $(c_i) = (R_i/2_p)]$, where '*i*' is the scheduling step and '*p*' is number of processors. 'R_i' is to be decreased at each step by $[R_{i+1}=R_i-pc_i]$. Aiken et al. [30] has proposed a new loop parallelization technique called 'Perfect Pipelining'. Loops are not handled in an efficient manner by parallelizing compilers. Irregular parallelism is captured by fine-grain transformations inside a loop body which is not agreeable to coarser approaches. Irregular forms of parallelism are sacrificed by coarse methods in favour of pipelining iterations. The authors have presented a new transformation in this paper 'Perfect Pipelining', which is able to bridge the gap between these fine-grain and coarse-grain transformations and at the same time retaining the desirable features of both. The desirable features are retained even in the presence of resource constraints as well as conditional branches. Polychronopoulos et al. [31] proposed 'guided self-scheduling' which is a new approach for scheduling arbitrarily nested parallel program loops on shared memory multiprocessor systems. The most important factor to achieve high system as well as program performance is to utilize loop parallelism. Guided self-scheduling is well suited to be implemented on real parallel machines because of its simplicity. Minimal synchronization overhead as well as load balancing are the two most important objectives which are achieved simultaneously by guided self-scheduling. Its parameterized nature that permits us for tuning it for different systems as well as are its insensitivity to the initial processor configuration are the two other interesting properties of guided self-scheduling. Kruskal et al. [32] however proposed the simplest method to decide the chunk size. Authors calculate the optimal chunk size as *[chunk (c) =N/p]*, where '*N*' is the number of processors, and '*p*' is the number of processors. This scheme has the minimal communication overhead for obvious reason as the no. of chunks are exactly equal to the no of processors, but it inhibits poor load balancing as the heterogeneity in the network increases.

All the frameworks that have been discussed here in the existing literature so far need to be programmed specifically. All of them need the knowledge of programming like advanced parallel language concepts and the end users in many cases find it very difficult to install and configure them. A few of them need very high end underlying resources to run. The discussed frameworks work on some particular load balancing problem and provide a solution for only specific type of problems. Most of them work only on multiprocessing systems. All these challenges combine to form the main source of motivation for the proposed framework. All the difficulties that have been discussed, are addressed in the proposed framework SIPS. The proposed framework can compute the given problem on five different scheduling algorithms i.e., chunk scheduling, guided self-scheduling (GSS), factoring, trapezoid self-scheduling (TSS) and, quadratic self-scheduling (QSS).

THE PROPOSED FRAMEWORK AND METHODOLOGY

The Proposed framework, Serial Algorithms in Parallel Systems (SIPS) follows the MASTER/SLAVE(s) architectures to solve the computation. Master (node) divides the workload between various slaves (nodes) available in the network. Each node computes the request and data given by MASTER node and send back results to the master node. In the initial stage, we extract all information from source code using AST (Abstract Syntax Tree), then it targets the loop where maximum parallelism can be achieved and distribute similar copy of code to all slaves (nodes) but with different for loop initialization and comparison conditions, so that maximum parallelism can be achieved and each node execute independent operation on each iteration. For parallelism the selection of loop is done manually and then phase of code conversion is performed to make it compatible to run on different slaves (nodes) and collect results. Each node then executes the code (local copy of code with modified loop) and sends results to the main node from where actual request for executing the code is initially made. The whole process can be represented using algorithm1 and figure1. In this whole process, data dependence plays an important role. High independence between different blocks of code results in higher parallelism, hence higher efficiency and low total execution time.

1. *Procedure SIPS* (L_n, N_n)
2. *START*
3. *Input Source Code*
4. *Generate Abstract Syntax Tree*
5. *Select the Target Loop*
6. *Analyse data dependence in the code before the Loop Between the elements used in the program and elements to be used in the Loop only.*
7. *Emulate the data dependence to assign values to effecting elements (like variables).*
8. *Get List of Live Nodes and select the nodes to be used for execution.*
9. *Divide Loop among live nodes with separate lower and upper bounds to achieve parallelism.*
10. *Start execution of code on each selected node.*
11. *Send back results to main node during execution or after execution as per applicability or requirement.*
12. *Merge the results on Main node.*
13. *EXIT*
14. *End Procedure.*

Algoritm1: SIPS Framework

The individual components of SIPS framework (Figure 1) and their purpose is discuss below:

Source Code

SIPS framework supports the normal Java code compatible with JDK 1.5. As Java is a strongly typed language, this framework follows strict naming and typing rules of Java. Using similar names for different variables can make things complex for this framework in some cases, so it is recommended to use different names for each element while coding.

Parser

Parser tool is used to extract all the information from source by converting source code into 'Abstract Syntax Tree' and then by visiting useful nodes in the tree. For example, all the info regarding loops i.e., Initial condition, comparison, update variable, body etc., can be extracted.

Abstract Syntax Tree

Abstract Syntax Tree (AST) [34] helps in exploiting parallelism available in loops, as it can extract useful information about the elements present in the syntax by parsing the source code. The parsed information can be converted back to the source code by un-parsing, which is helpful in delivering the modified code compatible with the parallel system.

It helps in representing the original code written in a programming language, in its Abstract Syntax Structure of original code written in a programming language. Every node of the tree denotes a construct occurring in the source code. The syntax does not represent every detail appearing in the real syntax of the source program. Grouping Bracket are implicit in the tree structure, and a syntactic construct like an if-condition-then expression is denoted by means of a single node with two branches.

The parser builds the abstract syntax trees when the source code translation is done by the compiler process. After the AST is built, more information is added to the AST by means of subsequent processing, e.g., semantic analysis (compilers) as shown in figure 2. Program analysis and program transformation systems can also AST.

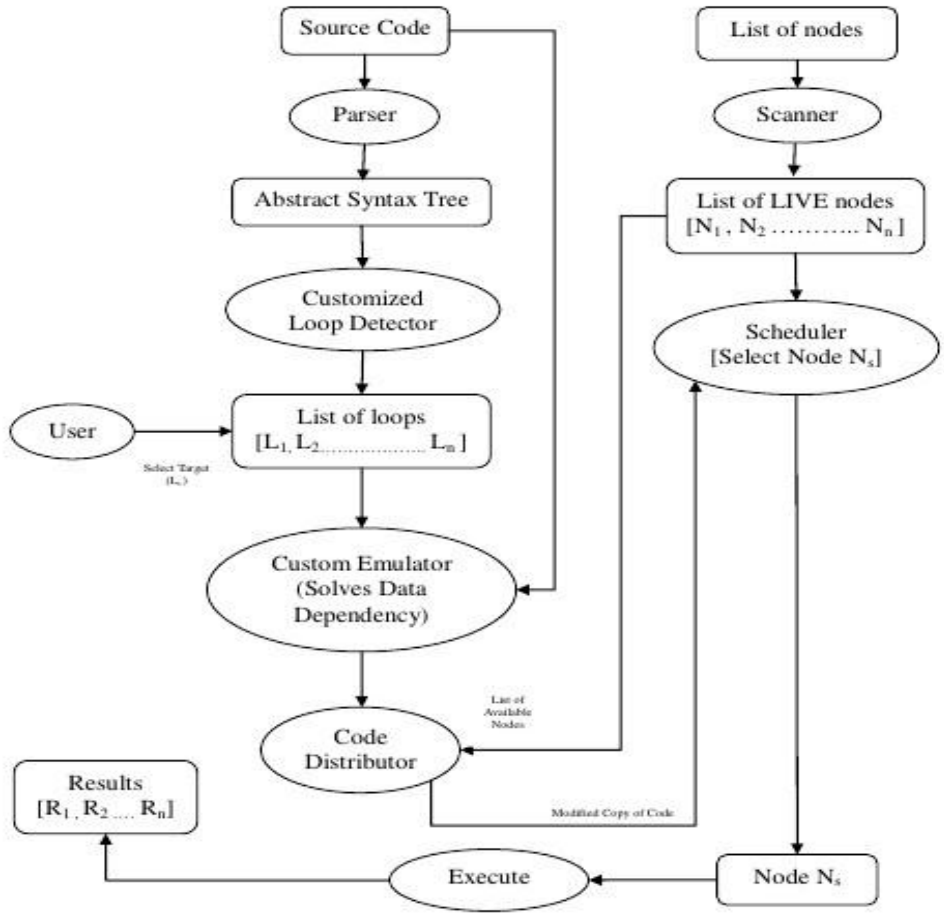


Figure 1: Architecture of SIPS framework

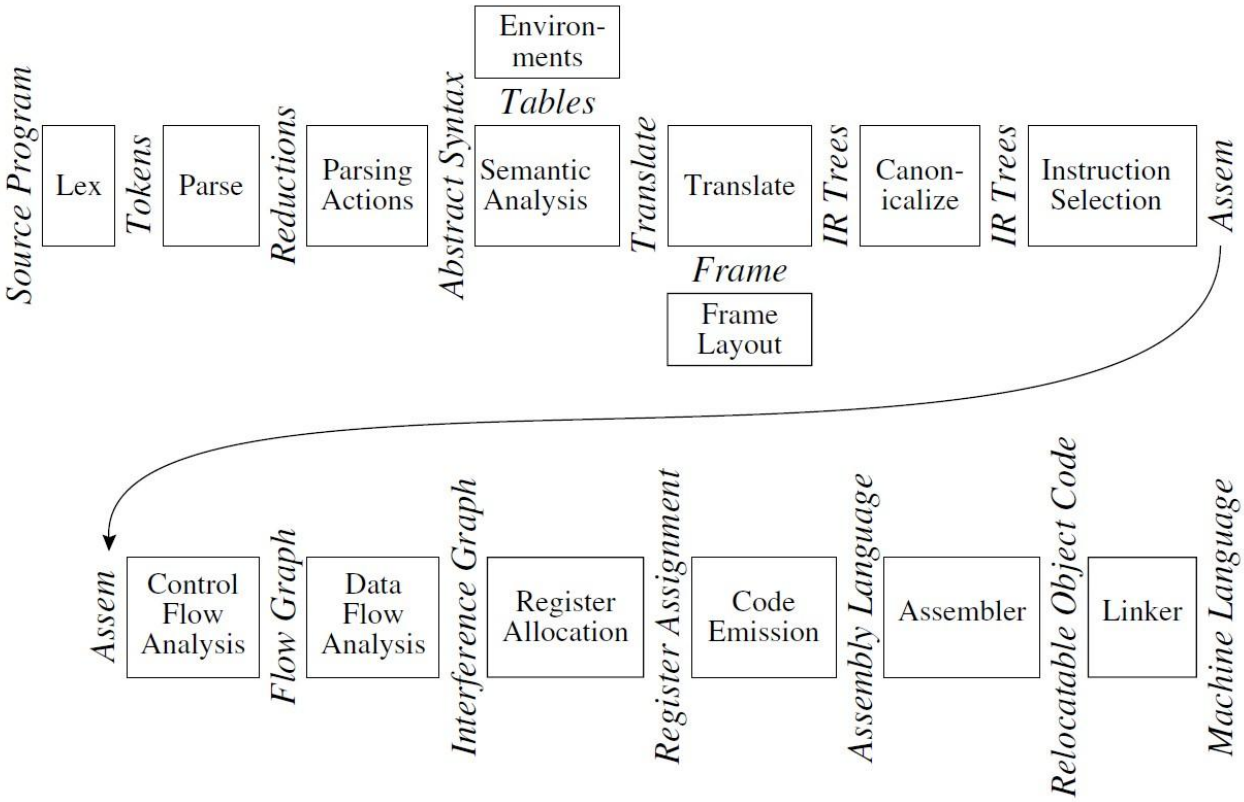


Figure 2: Phases of Compiler, and interfaces between them, W.A. Andrew et al. (2002)

Customised Loop Detector

It detects all the loops available in the source code, detecting even the hierarchy of loops. It then helps user to choose loops which can be targeted to achieve parallelism. User selects the loop(s) to be targeted and the framework pro- cesses this information in next phase of code emulation.

Custom Emulator

It takes source code and information from previous phases as input. It checks the data dependence between various elements of the program from source code with the help of AST and API(s) of framework. It helps Distributor to set Lower Bound and Upper Bound for each copy of loop(s). It also helps to generate consistent data to be used by all participating Nodes.

Scheduler

It picks up a live node from the list of live nodes selecting the live node on several basis such as availability, job queue, FCFS, round-robin or any other scheduling technique and then sends them a modified copy of source code to execute.

Code Distributor

It has a list of available nodes and it sets Lower bound and upper Bound for each loop and generates a modified copy of source code. Then this modified copy of source is passed on to the scheduler which carries out the selection process of a node that will be used for the execution of the modified source code.

Scanner

It scans the given List of nodes for the available live nodes in real time and also maintains the list of live nodes. If any node from List of live nodes gets into Down-time, scanner is responsible for reflecting the change in both lists.

Node(s)

As shown in figure1 for single process, Node is the slave part of the architecture which simply receive a part of code, execute it and send back the results to master. But if node submit a job in the system it will gain the properties of a master.

Inputs

This framework depends on number of inputs like source code, list of available nodes and selection of loop to achieve maximum parallelism.

Outputs

The main outputs we can obtain from the framework are list of live nodes and results from all the nodes that have participated in the job execution.

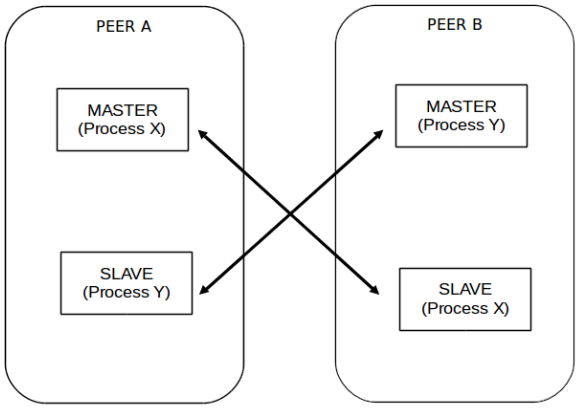


Figure 3: P2P Execution of two jobs at the same time using SIPS

As we can see from Figure 3, the SIPS framework supports peer-to-peer execution, where each peer has equal probability to become the master and all the other peers participating in the system will act as slaves for the job submitted by another node. In the fig.3, Peer 'A' acts as master for Process 'X' and Peer 'B' act as slave for the same Process 'X' whereas Peer 'B' acts as master for Process 'Y' and Peer 'A' act as slave for the same Process 'Y'.

The proposed framework is also capable of executing jobs in the master –slave fashion as can be seen in Figure 4, where Node 1 is acting as master and all other nodes are acting as slaves. The node from where the task is submitted is considered master and the selected nodes, responsible for the execution of task are considered to be slave nodes.

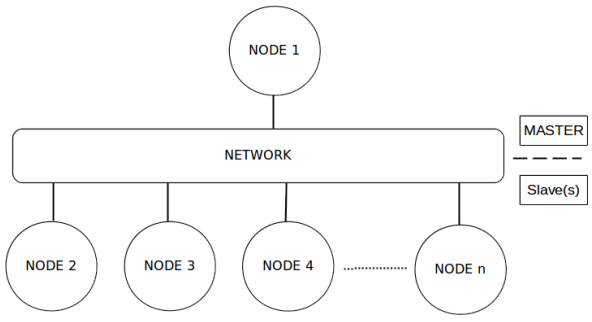


Figure 4: Master-Slave model

EXPERIMENTS AND ANALYSIS OF RESULTS

Two different networks are chosen for performing the tests on SIPS and evaluating its performance. The tests are named as Experiment-I and Experiment-II. In both the tests, the matrix multiplication problem has been taken as benchmark for performance evaluation of SIPS which is the standard benchmark program for the testing and evaluation of parallel computing frameworks. Comparisons based on NOH (Network Overhead), POH (Parsing Overhead) and TET (Total Execution Time) have been made for the different scheduling techniques of SIPS and JPPF for the problem size of 1000*1000(case I) and then for 2500*2500(case II). Finally, the performance evaluation is tested on the problem size of 5000*5000 where JPPF is unable to perform (Goes non responsive) while SIPS continued to run for 5000*5000 and 7500*7500 as well.

Under Experiment-I, the SIPS framework tests are carried out in a shared network environment consisting of a group of nodes. All of the salve nodes in the setup have 4 GB of RAM and Intel i3 first generation processors clocked at 3.2 GHz. Master Node used in our setup has 8 GB of RAM and Intel i7 second generation processor. The JPPF has been tested for 'rl' scheduling which is available in JPPF, as it was found to be the best performing scheduling technique of JPPF under the test environment.

In Experiment-II, a relatively dedicated environment comprising nodes having more capability is taken to perform the same test. The nodes belonging to the family of i3 quad core processors of 7th generation with 4 GB RAM were considered for the experiment. Based on availability, the master node considered belongs to the family of processors of i5 series quad-core processors with 3 GB RAM. First, the 'rl' scheduling of JPPF is tested for the problem size of 1000*1000 and further for 2500*2500, but for 2500*2500 the processing time increased to a great extent. So, customized scheduling was tried for JPPF for having best performance (we use name JPPF-chunk here). Where the number of threads for a particular number of nodes is kept three times the number of nodes for the reason that each node has four processors, the one processor was considered and left free for running the client program deliberately. This actually became very close to the quadratic self-scheduling of SIPS where the delta values were chosen for creating chunks as many as equal to three times the number of nodes. Specifically because of this reason, JPPF performed better, as in JPPF there is no parsing overhead as shown in figure [10].

The various formulas on the basis of which both the frameworks have been tested are given as following:

Parsing Overhead

In the proposed framework, all programs needs to be parsed before execution to detect available loops in the code. But parsing overhead is the time taken by framework to read the code and modify it, so that it can be executed on different Nodes present in the system. It remains almost constant for same code.

POH (Framework) = {ModificationEndTimeStamp – ParsingStartTimeStamp} (4.1)

Network/Communication Overhead

It is the communication delay which occurs because of network traffic. Network overhead (NOH) increases with the increasing number of nodes or increasing number of chunks, as it requires more communication between master and slave nodes in the system. Due to this, NOH plays a major role in total execution time of a job. It can be measured using following formula:

NOH (n) = {Execution time recorded by master for Node n – Execution Time recorded by node n}

(4.2)

Scheduling Overhead

It can be defined as the time taken by master to distribute the task among other nodes. It directly depends on master node's performance. It also includes parsing overhead.

SOH (framework) = {Scheduling End Time Stamp – Start Time Stamp}

(4.3)

Total Execution Time

Total Execution Time (TET) id the actual time taken to execute the job by framework on all nodes. It is the total time from submission of the job till the last calculated chunk is received by at the master node. Total execution is also a function of NOH (Equation 4.2). It is calculated on the following formula:

TET (framework) = {End Time Stamp (master) – Start Time Stamp (master)}

(4.4)

Figure 5 shows result of very initial test of SIPS frame work. The system is run to execute the simple for loop program to print 1000000 numbers on screen and the graph shows the total execution time against the selected number of nodes. It is clear from the graph that as the no. of nodes are increased, the total execution time reduces exponentially. Though TET almost stabilizes as the no. of nodes (n) reaches 7. This initial test of SIPS proved that the proposed framework follows the general trend of parallel computing.

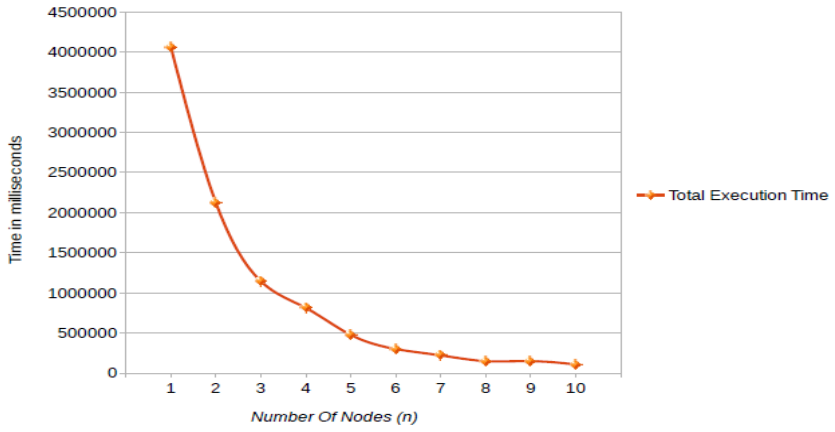


Figure 5: For Loop Test of SIPS

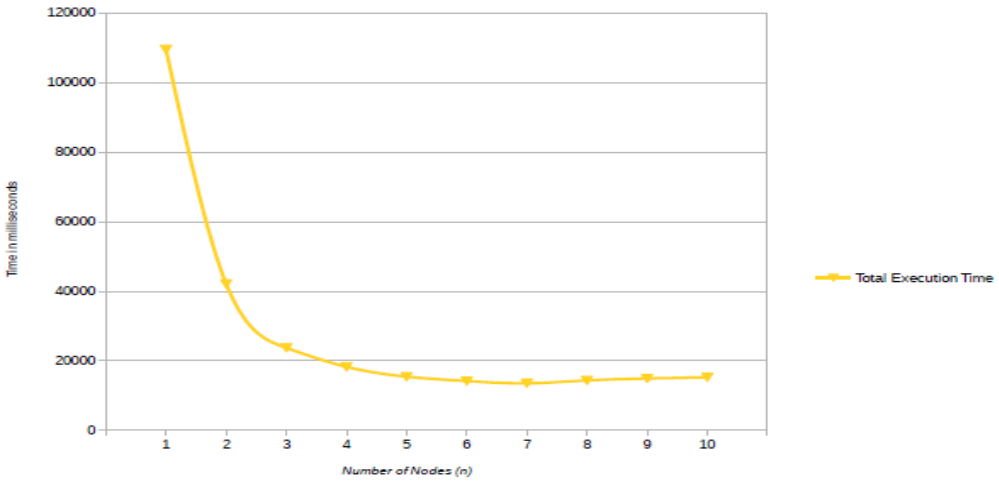


Figure 6: Matrix Multiplications (1000*1000) testing

Figure 6 shows the result of next test run of SIPS, where the system is run to execute a 2-D matrix multiplication program of matrix size 1000*1000. This graph also show the similar results as shown in figure 5. In this run, the time exponentially reduces till node 7 but there after it tends to increase a little as the number of nodes are increased. This is due to increase in the communication overhead. Both the initial tests of SIPS show that, it follows the general trend of parallel computing.

Figure 7 shows the results of experiment-I (case-I), where the SIPS is run to execute the benchmark 1000*1000 matrix multiplication problem for all the scheduling i.e. chunk scheduling, Guided Self scheduling, Factoring, Trapezoid Self Scheduling, Quadratic Self scheduling and is compared with one of the existing parallel computing frame work JPPF (Java Parallel Programming Framework), with its existing 'rl' load balancing technique. The graph is plotted for increasing number of nodes vs the total execution time (TET). It can be seen from the graph that all the scheduling of SIPS touch the minimum execution times around 6-7 nodes and beyond that the execution time increases due to increased network overhead. The graph shows that SIPS is more consistent in following the general trend of parallel computing whereas the JPPF's 'rl' scheduling is a little inconsistent in the middle (from 7 nodes to 13 nodes) in this experiment. The performance of simple chunk scheduling is best due to least communication overhead.

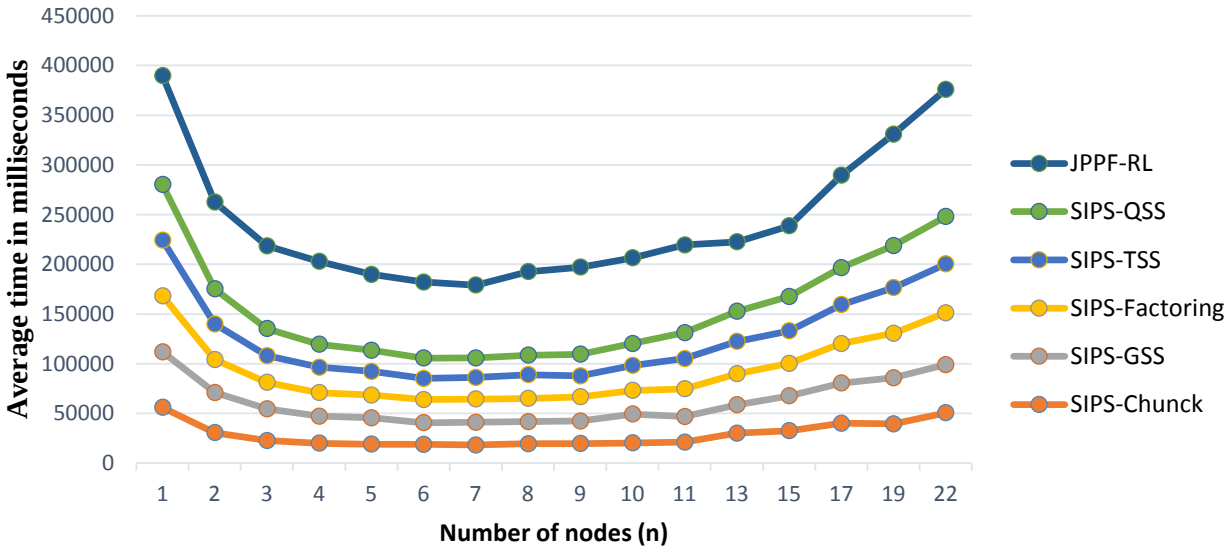


Figure 7: EXPERIMENT I (Case-I) - 1000*1000 Matrix Multiplication Problem

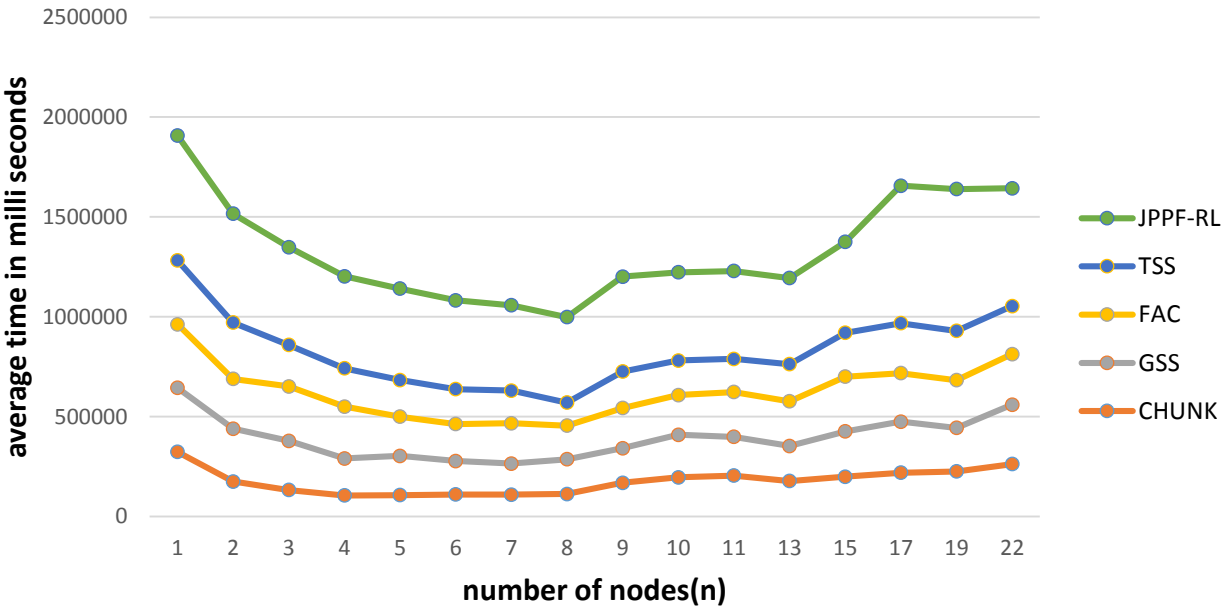


Figure 8: EXPERIMENT I (Case-II) - 2500*2500 Matrix Multiplication Problem

The results of experiment-I (case-II) are shown in Figure 8. This experiment is performed to test and compare the performance of SIPS (under various scheduling) and JPPF with the increased load. The 2500*2500 matrix multiplication problem for all the scheduling i.e. chunk scheduling, Guided Self scheduling, Factoring, and Trapezoid Self Scheduling is compared with one of the existing parallel computing frame work JPPF(Java Parallel Programming Framework), with its existing 'rl' load balancing technique. In this experiment the QSS is not included because the choice of optimal delta value becomes very difficult to calculate for the 2500*2500 matrix manually. Though in experiment-II a brute force program is made and inducted in SIPS to calculate the optimal delta value for the given number of nodes. The graph is plotted between 'number of nodes' and the total execution time. It can be seen from the graph that all the scheduling of SIPS touch the minimum execution times around 6-7 nodes and beyond that the execution time increases due to increased network overhead. Though, all the scheduling in SIPS are visibly performing better than the 'rl' scheduling of JPPF. The graph also show that SIPS is consistently following the trend of parallel computing better than the JPPF in this experiment with the additional claims of SIPS that is reduction in the programing effort and time at the programmer's end. Again the chunk scheduling of SIPS shows the best performance and clearly JPPF's 'rl' scheduler takes more time.

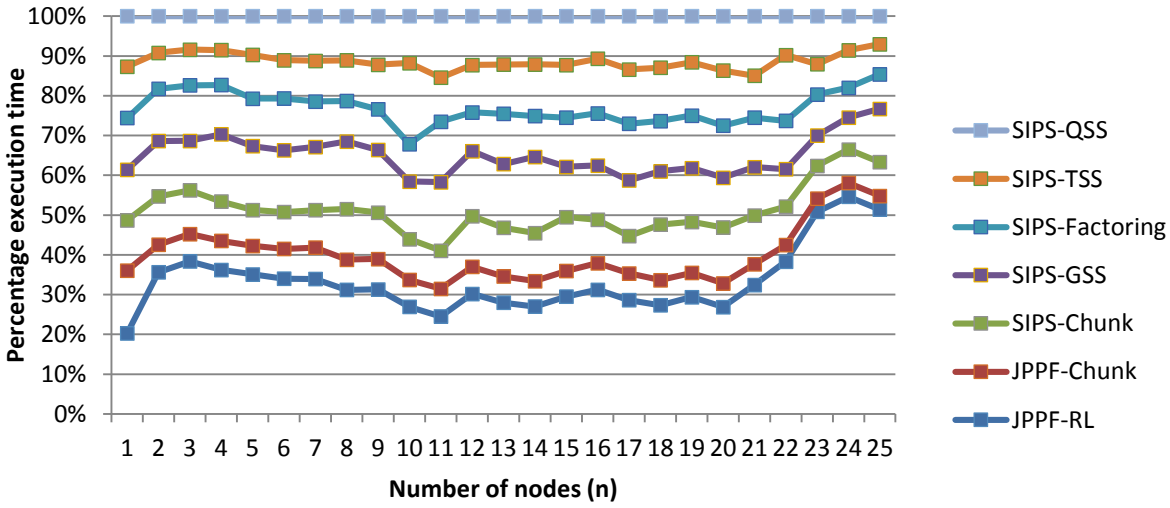
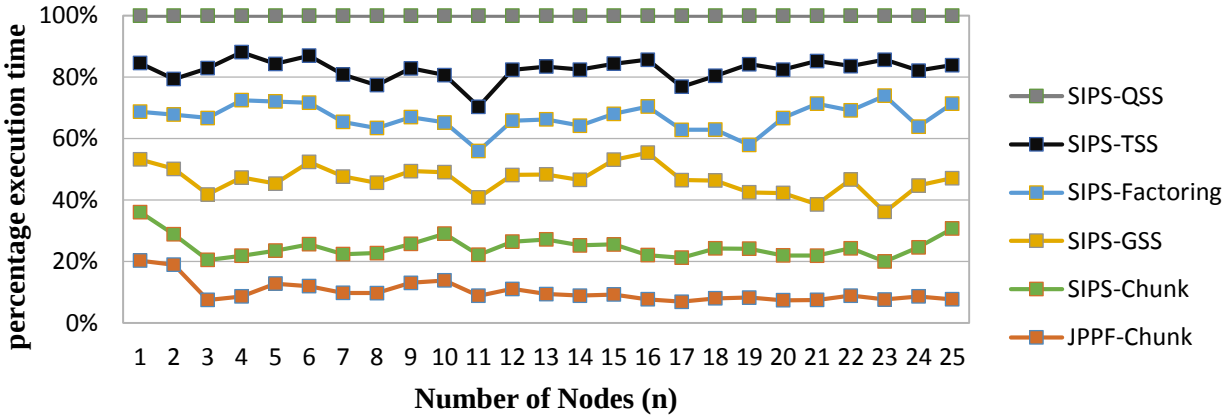


Figure 9: EXPERIMENT II (Case-I) - 1000*1000 Matrix Multiplication Problem



.

Figure 10: EXPERIMENT II (Case-II) - 2500*2500 Matrix Multiplication Problem

The experiment-II is done primarily to test the proposed framework under almost dedicated network with more capable nodes (as explained earlier). Figure 9 shows the results of experiment-II, case-I. As it is a nearly dedicated network with better capability nodes, the execution time tends to decrease faster and reaches minimum at 3-4 nodes for nearly all the scheduling techniques and then becomes almost stable till the 18-20 nodes. This shows that the communication overhead have less effect as the no. of nodes increase in the system which is due to the better availability of network and more

capable slave nodes. It can be seen from the results that under this type of environment the different scheduling techniques tends to behave differently. Clearly JPPF's 'rl' and 'chunk' scheduling are turning out to be best techniques respectively and the next best is 'chunk' scheduling implemented in SIPS.

Figure 10 is again an extension to the experiment-II to run the same experiment for the bigger problem 2500*2500 matrix multiplication program (case - II) to test the performance of various scheduling techniques in SIPS and JPPF under the increased load. JPPF's 'rl' load balancing technique was not included in this figure, because it was taking too much time (almost non responsive), for this specific test environment. This graph also shows the similar trends as seen in experiment-II case-I, where JPPF-chunk (customised for best performance in test environment) has shown better results due to the no parsing overhead incurred in JPPF

CONCLUSION

The proposed framework SIPS, is capable of parallelising any sequential algorithm with for loops. Though working with SIPS, the user needs to be careful in selecting the 'for' loop which is to be parallelized. The implementation of the code is done in JAVA, which is a platform independent language, thus makes the working of SIPS more user-friendly. Moreover, unlike the frameworks that have been discussed in the existing literature, SIPS does not need to be programmed specifically. SIPS works on loosely coupled and distributed environment, hence it is able to makes use of the available hardware. In this way, a lot of cost is reduced as the existing technology is tapped effectively. In addition to all this, SIPS is capable of working on different scheduling techniques and performs load balancing efficiently.

Results of experiment-I and experiment-II have shown that the proposed framework performs better than the existing scheduling techniques of JPPF, when run under different scheduling techniques. Results have also shown that SIPS performed better in shared environment, and when it comes to relatively dedicated environment, JPPF can be customised to perform better. SIPS works on standard scheduling techniques, and number of chunks are only calculated from formulae of those techniques. Though customized scheduling can be proposed in SIPS as well, to match the underlying network conditions. Despite some negatives, SIPS has inhibited more stability in performance and seems more suitable for shared networks (which is the actual situation of most of the available networks), and with its additional claims of reduced programming efforts, SIPS does have its advantages over other frameworks. Easy to install, program and use, SIPS can be seen as a potential parallel computing framework in its category.

REFERENCES

[1] Tournier, P. H., Bonazzoli, M., Dolean, V., Rapetti, F., Hecht, F., Nataf, F., ... & Darbas, M. (2017). Numerical Modeling and High-Speed Parallel Computing: New Perspectives on Tomographic Microwave Imaging for Brain Stroke Detection and Monitoring. *IEEE Antennas and Propagation Magazine*, *59*(5), 98-110.

[2] Wang, J., Rubin, N., Sidelnik, A., & Yalamanchili, S. (2016). LaPerm: Locality aware scheduler for dynamic parallelism on GPUs. *ACM SIGARCH Computer Architecture News*, *44*(3), 583-595.

[3] Jolivet, P., & Nataf, F. (2014). Hpddm: High-Performance Unified framework for Domain Decomposition methods, MPI-C++ library.

[4] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., & Stoica, I. (2014, October). GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI* (Vol. 14, pp. 599-613).

[5] Poulson, J., Marker, B., Van de Geijn, R. A., Hammond, J. R., & Romero, N. A. (2013). Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software (TOMS)*, *39*(2), 13.

[6] Ou, Y., Li, B., Yuan, Z., Hao, Q., Luan, Z., & Qian, D. (2012, December). LPFSC: A Light Weight Parallel Framework for Super Computing. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on* (pp. 453-458). IEEE.

[7] Zhang, Y., Gao, Q., Gao, L., & Wang, C. (2012). imapreduce: A distributed computing framework for iterative computation. *Journal of Grid Computing*, *10*(1), 47-68.

[8] Espeland, H., Beskow, P. B., Stensland, H. K., Olsen, P. N., Kristoffersen, S., Griwodz, C., & Halvorsen, P. (2011, September). P2G: A framework for distributed real-time processing of multimedia data. In *Parallel Processing Workshops (ICPPW), 2011 40th International Conference on* (pp. 416-426). IEEE.

[9] Zhou, Y., Chiu, D. M., & Lui, J. C. (2011). A simple model for chunk-scheduling strategies in P2P streaming. *IEEE/ACM Transactions on Networking*, *19*(1), 42-54.

[10] Honjo, M., Kubota, A., & Kitamura, T. (2010, November). Parallel programming framework for heterogeneous computing environment with Xen virtualization. In *TENCON 2010-2010 IEEE Region 10 Conference* (pp. 1100-1105). IEEE.

[11] Jose, L., de Souza, S. M. A., & Foltran Jr, D. C. (2010, October). Towards a peer-to-peer framework for parallel and distributed computing. In *Computer Architecture and High Performance Computing*

(SBAC-PAD), 2010 22nd International Symposium on (pp. 127-134). IEEE.

[12] Ma, Y., Liu, D., & Li, J. (2009, July). A new framework of cluster-based parallel processing system for high-performance geo-computing. In *Geoscience and Remote Sensing Symposium, 2009 IEEE International, IGARSS 2009* (Vol. 4, pp. IV-49). Ieee.

[13] Chiang, S. H., & Vasupongayya, S. (2008). Design and potential performance of goal-oriented job scheduling policies for parallel computer workloads. *IEEE Transactions on Parallel and distributed systems*, *19*(12), 1642-1656.

[14] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P. K., & Currey, J. (2008, December). DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI* (Vol. 8, pp. 1-14).

[15] Chen, C., & Tsai, K. C. (2008). The server reassignment problem for load balancing in structured P2P systems. *IEEE Transactions on Parallel and Distributed Systems*, *19*(2), 234-246.

[16] Lindholm, E., Nickolls, J., Oberman, S., & Montrym, J. (2008). NVIDIA Tesla: A unified graphics and computing architecture. *IEEE micro*, *28*(2).

[17] Hecht, F. (2007). *FreeFem++: version 2.21*. Laboratoire Jacques-Louis Lions.

[18] Diaz, J., Reyes, S., Nino, A., & Munoz-Caro, C. (2006, September). A Quadratic Self-Scheduling Algorithm for Heterogeneous Distributed Computing Systems. In *Cluster Computing, 2006 IEEE International Conference on* (pp. 1-8). IEEE.

[19] Goodale, T., Allen, G., Lanfermann, G., Massó, J., Radke, T., Seidel, E., & Shalf, J. (2002, June). The cactus framework and toolkit: Design and applications. In *International Conference on High Performance Computing for Computational Science* (pp. 197-227). Springer, Berlin, Heidelberg.

[20] Goodale, Tom. (2002). "The Cactus Framework and Toolkit: Design and Applications, Vector and Parallel Processing-5th International Conference." *Lecture Notes in Computer Science*.

[21] Andrew, W. A., & Jens, P. (2002). Modern compiler implementation in Java.

[22] Sorasen, O., & Lundh, Y. (1997, February). Swipp-a multicomputer framework for bulk synchronous parallel computing. In *Performance, Computing, and Communications Conference, 1997. IPCCC 1997., IEEE International* (pp. 108-114). IEEE.

[23] Foster, I., & Kesselman, C. (1997). Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, *11*(2), 115-128.

[24] Lilja, D. J. (1994). Exploiting the parallelism available in loops. *Computer*, *27*(2), 13-26.

[25] Markatos, E. P., & LeBlanc, T. J. (1994). Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed systems*, *5*(4), 379-400.

[26] Tzen, T. H., & Ni, L. M. (1993). Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on parallel and distributed systems*, *4*(1), 87-98.

[27] Yau, S. S., Bae, D. H., & Chidambaram, M. (1992, April). A framework for software development for distributed parallel computing systems. In *Distributed Computing Systems, 1992., Proceedings of the Third Workshop on Future Trends of* (pp. 240-246). IEEE.

[28] Hummel, S. F., Schonberg, E., & Flynn, L. E. (1992). Factoring: A method for scheduling parallel loops. *Communications of the ACM*, *35*(8), 90-101.

[29] Sunderam, V. S. (1990). PVM: A framework for parallel distributed computing. *Concurrency and Computation: Practice and Experience*, *2*(4), 315-339.

[30] Aiken, A., & Nicolau, A. (1988, March). Perfect pipelining: A new loop parallelization technique. In *European Symposium on Programming* (pp. 221-235). Springer, Berlin, Heidelberg.

[31] Polychronopoulos, C. D., & Kuck, D. J. (1987). Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *Ieee transactions on computers*, *100*(12), 1425-1439.

[32] Kruskal, C. P., & Weiss, A. (1985). Allocating independent subtasks on parallel processors. *IEEE Transactions on Software engineering*, (10), 1001-1016.

[33] (2011)The JPPF website [online]. Available: http://jppf.org/about.php

[34] Nakayama, K., & Sakai, E. (2014, June). Source code pattern as anchored abstract syntax tree. In *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on* (pp. 170-173). IEEE.