Mississippi State University
Scholars Junction

Theses and Dissertations Theses and Dissertations

5-10-2003

# Performance Analysis and Evaluation of Dynamic Loop Scheduling Techniques in a Competitive Runtime Environment for Distributed Memory Architectures

Mahadevan Balasubramaniam

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation
Balasubramaniam, Mahadevan, "Performance Analysis and Evaluation of Dynamic Loop Scheduling Techniques in a Competitive Runtime Environment for Distributed Memory Architectures" (2003). *Theses and Dissertations*. 3495.
https://scholarsjunction.msstate.edu/td/3495

This Graduate Thesis - Open Access is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.

# PERFORMANCE ANALYSIS AND EVALUATION OF DYNAMIC LOOP SCHEDULING TECHNIQUES IN A COMPETITIVE RUNTIME ENVIRONMENT FOR DISTRIBUTED MEMORY ARCHITECTURES

By

Mahadevan Balasubramaniam

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science and Engineering
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

May 2003

Copyright by

Mahadevan Balasubramaniam

2003

PERFORMANCE ANALYSIS AND EVALUATION OF DYNAMIC LOOP SCHEDULING TECHNIQUES IN A COMPETITIVE RUNTIME ENVIRONMENT FOR DISTRIBUTED MEMORY ARCHITECTURES

By

Mahadevan Balasubramaniam

Approved:

Ioana Banicescu
Associate Professor of Computer Science
and Engineering
(Major Advisor)

Thomas Philip
Professor of Computer Science
and Engineering
(Committee Member)

Joerg Meyer
Adjunct Assistant Professor of Computer Science
and Engineering
(Committee Member)

Susan M. Bridges
Professor of Computer Science
and Engineering
and Graduate Coordinator

A. Wayne Bennett
Dean of the College of Engineering

Name: Mahadevan Balasubramaniam

Date of Degree: May 10, 2003

Institution: Mississippi State University

Major Field: Computer Science and Engineering

Major Professor: Dr. Ioana Banicescu

Title of Study: PERFORMANCE ANALYSIS AND EVALUATION OF DYNAMIC LOOP SCHEDULING TECHNIQUES IN A COMPETITIVE RUNTIME ENVIRONMENT FOR DISTRIBUTED MEMORY ARCHITECTURES

Pages in Study: 107

Candidate for Degree of Master of Science

Parallel computing offers immense potential to solve large, complex scientific problems. Load imbalance is a major impediment in obtaining high performance by a parallel system. One principal form of parallelism found in scientific applications is data parallelism. Loops without dependencies are data parallel. During the execution of large parallel loops, computational requirements vary due to problem, algorithmic and systemic characteristics. These factors lead to load imbalance which in turn degrades the performance of an application. Over the years, a number of dynamic loop scheduling techniques have been proposed to address one or more of these factors. However, there is no single strategy that works well for different problem domains and system characteristics. Moreover, load balancing during runtime is complicated because of its need for dynamic data redistribution. Therefore, there is a distinct need to integrate the dynamic loop scheduling techniques into a single package and provide them as an application programming interface (API) to the application developer. In recent years, along this direction, a number of dynamic loop scheduling techniques have been integrated into the compiler technologies for shared memory environments. On the other hand, there is no such integrated approach for distributed memory applications. The purpose of this thesis is to present the design, implementation and effectiveness of an integrated approach:the dynamic loop scheduling techniques are integrated

into a runtime system for distributed memory architectures. For this purpose, we choose the newly developed parallel runtime environment for multicomputer architecture (PREMA) with its main components: the data movement and control substrate (DMCS) and mobile object layer (MOL). This runtime system has recently been developed and has demonstrated to be one of the most competitive runtime systems for distributed memory architectures.

The significance of this work is that the proposed API will enhance the performance of parallel applications by reducing the load imbalance among processors caused by a wide range of factors and will reduce the software developmental cost required for load balancing. With the integration of the scheduling capabilities into the runtime system, its applicability has been expanded. The performance of the API has been evaluated qualitatively and quantitatively. The overhead of the API has been studied analytically and measured experimentally. Three parallel benchmarks including scientific applications of general interest ($N$-body simulations, automatic quadrature routine and unstructured grid heat solver) were considered for experimentation purpose. Based on the experiments conducted, a cost improvement of up to 76% over the straight forward parallel benchmark has been obtained. For certain application characteristics, the overhead of the runtime system was found to be within 10% of the underlying messaging layer. These results demonstrate that, in large scientific applications it is possible and desirable to combine the rich functionality of a runtime system with the advantages of scheduling techniques to achieve high performance.

## ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my major professor, Dr. Ioana Banicescu, for giving me an opportunity to work on research under her guidance and for her constant academic and moral support through the two years that we have known each other. I would like to thank Dr. Ricalindo Carino and Mr. Sheikh Ghafoor for their precious technical advice. I would also like to thank my committee, Dr. Thomas Philip and Dr. Joerg Meyer for their valuable contributions. Special thanks to the Engineering Research Center for providing the facilities needed for this research effort. Finally, a token of appreciation to all my friends who have made my stay at MSU an enjoyable one.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENT . . . . . . . . . . ii

LIST OF TABLES . . . . . . . . . . v

LIST OF FIGURES . . . . . . . . . . vi

NOMENCLATURE . . . . . . . . . . x

CHAPTER

I. INTRODUCTION . . . . . . . . . . 1
- 1.1 Related Work . . . . . . . . . . 3
- 1.2 Proposed Work . . . . . . . . . . 3
- 1.3 Problem Statement . . . . . . . . . . 4
- 1.4 Goals . . . . . . . . . . 5
- 1.5 Thesis Structure . . . . . . . . . . 5
- 1.6 Summary . . . . . . . . . . 6

II. LITERATURE REVIEW . . . . . . . . . . 7
- 2.1 Data parallelism and loop scheduling . . . . . . . . . . 7
  - 2.1.1 Static chunking and Self scheduling . . . . . . . . . . 9
  - 2.1.2 Fixed size chunking . . . . . . . . . . 10
  - 2.1.3 Guided Self Scheduling . . . . . . . . . . 10
  - 2.1.4 Factoring . . . . . . . . . . 11
  - 2.1.5 Weighted Factoring . . . . . . . . . . 12
  - 2.1.6 Adaptive Weighted Factoring . . . . . . . . . . 12
  - 2.1.7 Adaptive Factoring . . . . . . . . . . 13
  - 2.1.8 Fractiling . . . . . . . . . . 15
- 2.2 Runtime System . . . . . . . . . . 16
  - 2.2.1 Nexus . . . . . . . . . . 16
  - 2.2.2 Tulip . . . . . . . . . . 17
  - 2.2.3 C Region Library . . . . . . . . . . 17
  - 2.2.4 TreadMarks . . . . . . . . . . 17
  - 2.2.5 MOL/DMCS . . . . . . . . . . 17
- 2.3 Parallelizing C Compiler . . . . . . . . . . 18
- 2.4 Static partitioning . . . . . . . . . . 18
- 2.5 Dynamic partitioning . . . . . . . . . . 18

CHAPTER Page

2.6 Summary . . . . . 19

III. LIBRARY DESIGN . . . . . 21
3.1 One-sided vs Two-sided communication . . . . . 21
3.2 Parallel Runtime Environment . . . . . 22
3.2.1 Data Movement and Control Substrate . . . . . 23
3.2.2 Mobile Object Layer . . . . . 23
3.3 The DLBL library . . . . . 24
3.3.1 DLBL structure . . . . . 25
3.3.2 Phases of load balancing . . . . . 25
3.4 Memory Allocation Strategies . . . . . 26
3.4.1 Strategy I . . . . . 26
3.4.2 Strategy II . . . . . 28
3.5 Handlers . . . . . 29
3.6 Master-Slave Strategy for Two-Sided Model . . . . . 31
3.7 Master-Slave Strategy for One-Sided Model . . . . . 32
3.8 Summary . . . . . 35

IV. LIBRARY EVALUATION . . . . . 36
4.1 Modes of Evaluation . . . . . 36
4.2 Library overhead . . . . . 37
4.3 Experimental setup . . . . . 39
4.4 Performance Metrics . . . . . 39
4.4.1 Cost . . . . . 39
4.4.2 Effectiveness . . . . . 39
4.4.3 Coefficient of Variation . . . . . 39
4.5 $N$-body Simulations . . . . . 40
4.6 Automatic Quadrature Routine . . . . . 54
4.7 Heat Solver . . . . . 72
4.8 Summary . . . . . 78

V. OVERHEAD ANALYSIS . . . . . 80
5.1 Latency Measurements . . . . . 80
5.2 Experimental Evaluation . . . . . 84
5.2.1 $N$-body Simulations . . . . . 85
5.2.2 Automatic Quadrature Routine . . . . . 89
5.2.3 Heat Solver . . . . . 94
5.3 Analytical Evaluation . . . . . 96
5.4 Summary . . . . . 98

VI. CONCLUSIONS AND FUTURE WORK . . . . . 100
6.1 Accomplishments . . . . . 100
6.2 Lessons Learned . . . . . 102
6.3 Future Research Directions . . . . . 103
6.4 Summary . . . . . 104

REFERENCES . . . . . 105

LIST OF TABLES

TABLE Page

2.1 Chunk sizes for different scheduling schemes . . . . . . . . . . . . . . . . . . . . . . 16

4.1 Parallel application cost with and without the library . . . . . . . . . . . . . . . . 38

4.2 Best case C.O.V for $N$-body simulations I . . . . . . . . . . . . . . . . . . . . . . . 52

4.3 Best case C.O.V for $N$-body simulations II . . . . . . . . . . . . . . . . . . . . . . 52

4.4 % cost improvement over static chunking for $N$-body simulations . . . . . . . . . 53

4.5 Best case C.O.V for automatic quadrature routine I . . . . . . . . . . . . . . . . . 70

4.6 Best case C.O.V for automatic quadrature routine II . . . . . . . . . . . . . . . . . 70

4.7 % cost improvement over static chunking for automatic quadrature routine . . . 71

4.8 % cost improvement over static chunking for heat solver . . . . . . . . . . . . . . 73

## LIST OF FIGURES


| FIGURE | | Page |
|---|---|---|
| 2.1 | Data parallel computation | 8 |
| 2.2 | Morton ordering | 15 |
| 3.1 | Parallel runtime system architecture | 22 |
| 3.2 | DMCS Execution Model | 23 |
| 3.3 | DLBL architecture | 25 |
| 3.4 | Strategy I | 26 |
| 3.5 | Data movement | 27 |
| 3.6 | Buffer allocation | 28 |
| 3.7 | Strategy II | 28 |
| 3.8 | Data movement and storage | 29 |
| 3.9 | Master-Slave strategy for two-sided model | 31 |
| 3.10 | Master-Slave strategy for one-sided model | 32 |
| 4.1 | Different modes of evaluation | 36 |
| 4.2 | Sample DLBL application | 37 |
| 4.3 | Corner distribution | 42 |
| 4.4 | Gaussian distribution | 42 |
| 4.5 | Cost: Uniform-20k particles | 43 |
| 4.6 | Cost: Uniform-50k particles | 43 |
| 4.7 | Cost: Uniform-100k particles | 44 |
| 4.8 | Cost: Corner-20k particles | 44 |
| 4.9 | Cost: Corner-50k particles | 45 |
| 4.10 | Cost: Corner-100k particles | 45 |

FIGURE Page


4.11 Cost: Gaussian-20k particles . . . . . 46
4.12 Cost: Gaussian-50k particles . . . . . 46
4.13 Cost: Gaussian-100k particles . . . . . 47
4.14 Effectiveness: Uniform-20k particles . . . . . 47
4.15 Effectiveness: Uniform-50k particles . . . . . 48
4.16 Effectiveness: Uniform-100k particles . . . . . 48
4.17 Effectiveness: Corner-20k particles . . . . . 49
4.18 Effectiveness: Corner-50k particles . . . . . 49
4.19 Effectiveness: Corner-100k particles . . . . . 50
4.20 Effectiveness: Gaussian-20k particles . . . . . 50
4.21 Effectiveness: Gaussian-50k particles . . . . . 51
4.22 Effectiveness: Gaussian-100k particles . . . . . 51
4.23 Front distribution . . . . . 56
4.24 Back distribution . . . . . 56
4.25 Center distribution . . . . . 57
4.26 Scatter distribution . . . . . 57
4.27 Cost: 3780-Front . . . . . 58
4.28 Cost: 3780-Back . . . . . 58
4.29 Cost: 3780-Center . . . . . 59
4.30 Cost: 3780-Scatter . . . . . 59
4.31 Cost: 7560-Front . . . . . 60
4.32 Cost: 7560-Back . . . . . 60
4.33 Cost: 7560-Center . . . . . 61
4.34 Cost: 7560-Scatter . . . . . 61
4.35 Cost: 15120-Front . . . . . 62
4.36 Cost: 15120-Back . . . . . 62
4.37 Cost: 15120-Center . . . . . 63
4.38 Cost: 15120-Scatter . . . . . 63
4.39 Effectiveness: 3780-Front . . . . . 64
4.40 Effectiveness: 3780-Back . . . . . 64
4.41 Effectiveness: 3780-Center . . . . . 65

FIGURE Page


4.42 Effectiveness: 3780-Scatter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.43 Effectiveness: 7560-Front . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.44 Effectiveness: 7560-Back . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.45 Effectiveness: 7560-Center . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.46 Effectiveness: 7560-Scatter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.47 Effectiveness: 15120-Front . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.48 Effectiveness: 15120-Back . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.49 Effectiveness: 15120-Center . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.50 Effectiveness: 15120-Scatter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.51 Cylinder . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.52 Cost: 105K grid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.53 Cost: 148K grid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.54 Cost: 200K grid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.55 Cost: 300K grid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.56 Effectiveness: 105K grid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
4.57 Effectiveness: 148K grid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
4.58 Effectiveness: 200K grid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.59 Effectiveness: 300K grid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
5.1 Latency measurement using polls and requests . . . . . . . . . . . . . . . . . . . . 81
5.2 Latency measurement using sends and receives . . . . . . . . . . . . . . . . . . . . 81
5.3 Transfer time vs Message size with 0 sec wait time . . . . . . . . . . . . . . . . . 83
5.4 Transfer time vs Message size with 0.25 sec wait time . . . . . . . . . . . . . . . 83
5.5 Startup time vs Message size . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
5.6 $N$-body-FSC cost comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.7 $N$-body-GSS cost comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
5.8 $N$-body-FACT cost comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
5.9 % cost variation of $N$-body-FSC . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.10 % cost variation $N$-body-GSS . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.11 % cost variation $N$-body-FACT . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.12 AQR FSC cost comparison I . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
5.13 AQR FSC cost comparison II . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

FIGURE Page


5.14 AQR GSS cost comparison I . . . . . 90

5.15 AQR GSS cost comparison II . . . . . 91

5.16 AQR FACT cost comparison I . . . . . 91

5.17 AQR FACT cost comparison II . . . . . 92

5.18 % cost variation of AQR-FSC . . . . . 92

5.19 % cost variation AQR-GSS . . . . . 93

5.20 % cost variation AQR-FACT . . . . . 93

5.21 Heat Solver FSC cost comparison . . . . . 94

5.22 Heat Solver GSS cost comparison . . . . . 95

5.23 Heat Solver FACT cost comparison . . . . . 95

5.24 % cost variation for heat solver . . . . . 96

# NOMENCLATURE

Identifiers:

| | |
|---|---|
| API | Application Programming Interface |
| DLBL | Dynamic Load Balancing Library |
| PREMA | Parallel Runtime Environment for Multicomputer Architecture |
| MOL | Mobile Object Layer |
| DMCS | Data Movement and Control Substrate |
| MPI | Message Passing Interface |
| PVM | Parallel Virtual Machine |

# CHAPTER I

# INTRODUCTION

Parallel and distributed computing has been one of the fundamental research areas of computer science during the past couple of decades. The major motivation that spawned the birth of parallel computers is its ability to contribute towards finding solutions to large, complex problems in science and engineering. Scientific problems can often be decomposed into independent subproblems that can simultaneously be solved. Exploiting parallelism in the problem and simultaneously solving its independent subproblems is an important avenue to achieve considerable higher performance than those that are possible from a single processor. Thus, parallel computing gives the flexibility to solve very complex scientific problems like climate modeling, fluid turbulence, ocean circulation, etc., problems that would otherwise be impossible to solve on a uniprocessor machine.

All computers whether sequential or parallel, operate by executing instruction streams on data streams. Depending on how many instructions are executed on data streams, computers can be classified as follows: Single Instruction Single Data Stream (SISD), Single Instruction Multiple Data Stream (SIMD), Multiple Instruction Single Data Stream (MISD), and Multiple Instruction Multiple Data Stream (MIMD). One factor which typically influences parallel programming is the type of processor communication used. The way processors communicate is dependent on the type of memory architecture which can be classified as *shared memory* and *distributed memory.*

In a shared memory architecture, multiple processors operate in an independent fashion but all share the same memory resources. However, only one process can access a shared location at any given time. A race condition can occur when actions in two processes are not controlled and the behavior of the program depends on the order in which the actions happen. Therefore, the user has to explicitly synchronize access to the shared resource. Since all the memory accesses are handled through the shared bus, when many processors simultaneously access the shared resource, contention problems arise because the shared bus cannot handle all the requests for

memory retrieval at once. Hence, shared memory systems are difficult to scale as the number of processors increases.

In a distributed memory architecture, each processor has its own address space and operates in an independent manner. The processors are connected through the interconnection network. Data sharing across the communication network is done through *message passing*, a paradigm that is commonly used for programming distributed memory machines. Message passing can be done through several of its available application programming interfaces (APIs), such as Message Passing Interface (MPI) [1], Parallel Virtual Machine (PVM) [2] etc. Under this paradigm, all the communications are entirely handled by the user. The user must have an understanding of where the data resides and must determine when to communicate with other processors, what information to communicate, and which processors to communicate with.

The basic primitives in message passing are "send" and "receive" operations. These operations may be either synchronous or asynchronous. A synchronous operation will block until the operation is complete. An asynchronous operation will only initiate the operation and will not block. Operations are called "blocking" when send or receive buffers are free for reuse immediately after the call, and "non-blocking" when they are not. Therefore, a synchronous or an asynchronous operation is blocking or non-blocking respectively. On the other hand, all blocking or non-blocking calls do not operate in synchronous or asynchronous modes respectively. Asynchronous message passing allows some computation to be done while the message is still in transit, for instance when communication and computation can overlap. For a skillful programmer, this programming model can improve scalability as the number of processors increases.

It is only natural to associate high performance with parallel computing. However, there are several factors that affect the performance of an application running in a parallel and distributed environment like the choice of a parallel algorithm, load imbalance, type of interconnection network used, and others. The study of each of these factors and associated solutions have become through time, a major area of research. This study concentrates on improving the performance of applications running in a parallel and distributed environment through load balancing.

## 1.1 Related Work

Data parallel computations involve performing similar operations on different data objects simultaneously. In many scientific applications, loops without dependencies offer a rich source of data parallelism. Loop scheduling refers to ordering of execution of loop iterates onto processors. Proper loop scheduling leads to increased utilization of the system resources by reducing the load imbalance among the processors. Over the years, a number of dynamic loop scheduling algorithms have been proposed to address performance degradation caused by factors like: problem characteristics (i.e., non-uniform data distribution), algorithmic characteristics (i.e., condition statements), systemic characteristics (i.e., unpredictable data access latency, cache misses), etc. In general, the loop scheduling techniques range from static (i.e., static chunking) to more advanced dynamic scheduling techniques such as factoring [3], weighted factoring [4], adaptive factoring [5] and adaptive weighted factoring [6]. The advanced scheduling techniques are developed based on a probabilistic analysis, that dynamically computes assignments of variable-size chunks of loop iterates with variable execution times.

In the area of message passing, runtime systems have gained paramount importance because of its large impact on the performance and portability of the parallel application. Over the years, a number of runtime systems like Tulip [7], C region library [8], parallel runtime environment for multicomputer architecture (PREMA) [9, 10] have been developed. These runtime systems provide functionalities for process creation, process management, synchronization of process, etc.

In recent years, a number of dynamic loop scheduling techniques have been introduced into the compiler technology and have successfully been used in running parallel applications in shared memory environments. To the best of our knowledge, a combined approach of integrating dynamic loop scheduling techniques into a runtime system for distributed memory environments to address performance degradation of parallel applications due to load imbalance, has not been pursued yet.

## 1.2 Proposed Work

Despite the plethora of dynamic loop scheduling techniques presently developed, there is no single strategy that works well for different kinds of problems. Therefore, it is up to the user to select a technique that best satisfies the problem requirements, and integrate it into

the application. However, this approach introduces a new problem. The dynamic scheduling algorithm has to be integrated into the solution algorithm and thus, the user is faced with the responsibility and burden of maintaining both algorithms if he decides to improve the performance of the application through load balancing by himself. Even more great deal of effort is required from the application developer each time when a new load balancing method is integrated into the application.

The goal of this research work is to develop a new framework for load balancing scientific applications. Such a framework will be developed on top of an existing runtime system PREMA which supports features like one-sided communication, remote method invocation, remote read and write, globally distributed object space, etc. An application programming interface (API) will be developed to incorporate a number of dynamic scheduling techniques into a single package and to provide them to the user using a common interface. The API will be responsible for both, data migration and optimal data redistribution.

The user would initially implement an efficient and correct parallel solution algorithm which would ensure optimal decomposition (data-locality and communication) without no concern for load balancing. Once this phase is completed, the user can employ the API for dynamic load balancing the application with minimal modifications to the code. Thus, the user is spared from the problem of concurrently developing two very complex but conceptually independent components (the parallel solution algorithm and the dynamic scheduling algorithm) of an application.

### 1.3 Problem Statement

The research problem can be stated as follows:

1. An API that uses an integrated framework (dynamic loop scheduling-runtime system) for load balancing scientific applications on a distributed memory architecture is technically feasible.

2. When used by an application, the API will enhance its performance by reducing the load imbalance among the processors caused by a wide range of factors.

## 1.4 Goals

This goals of this thesis are to:

1. Study various dynamic loop scheduling techniques and runtime systems for distributed memory architectures to frame an in-depth understanding of the various issues in these research areas.

2. Implement a scalable, efficient and robust API for load balancing parallel adaptive applications.

3. Build the API, as a library package for general use.

4. Demonstrate that utilizing the new API for load balancing improves performance of a number of parallel applications.

5. Compare the performance of the various underlying scheduling techniques within the API for load balancing qualitatively and quantitatively.

6. Design guidelines for choosing an appropriate scheduling technique for an application depending on its characteristics.

7. Analyze analytically and experimentally, the performance of the API for load balancing with the other approaches to load balancing used by the present technology.

8. Draw some conclusions from the "lessons learnt" and propose further development from the experience gained.

## 1.5 Thesis Structure

The rest of this thesis is organized as follows: The first goal is addressed in chapter 1 by giving a detailed picture of the current status of dynamic loop scheduling techniques for scientific computing for distributed memory architectures as well as the recent advances in runtime systems. Goals 2 and 3 are answered in chapter 3. The design details of the proposed API are also presented in this chapter. Goals 4, 5 and 6 are satisfied in chapter 4. First, the experimental set up and the test applications are described. The presentation then continues to the interpretation of the

results. In chapter 5, the analytical and the experimental evaluation of the performance of the API with the other approaches to load balancing used by the present technology is presented to address goal 7. The final goal is achieved in chapter 6. The conclusions regarding the work accomplished along with the perspectives on the possible directions for future work are outlined in this chapter.

## 1.6 Summary

This chapter sets the stage for the development of this thesis by describing the motivation for this research work and briefly discussing the state-of-the-art in the related areas of research. The goals of this thesis are presented along with a number of objectives that have to be fulfilled in order to meet these goals. The significance and the impact of this research work are also emphasized in the context of the present technology.

# CHAPTER II

# LITERATURE REVIEW

In this chapter, an in depth survey of the related areas of research: (i) dynamic loop scheduling and (ii) runtime systems is attempted. Dynamic loop scheduling refers to ordering of execution of loop iterates onto processors during runtime. This is most often the choice because static scheduling uses iterative approach that do not seem to address load imbalances that occur during runtime. Runtime systems on the other hand have gained immense popularity because of their impact on the performance and portability of parallel applications. They provide functionalities for process creation, process management, process synchronization, etc.

This chapter is organized as follows. In section 2.1, the dynamic loop scheduling techniques for data parallel computations are briefly discussed. In section 2.2, the need for the runtime system is explained and some of the commonly available runtime systems are reviewed. An overview of sun ANSI/ISO C compiler is given in section 2.3. In section 2.4 and 2.5, a survey of some of the commonly used load balancing tools is presented.

## 2.1 Data parallelism and loop scheduling

Data parallel computations involve performing similar operations on different data objects simultaneously. In many scientific applications, loops without dependencies offer a significant source of data parallelism. Figure 2.1 represents a very simplified example of a data parallel computation. A loop construct can be in any language method that permits repetitive performance of a process. For example, it can be a *for* loop in a C like language, a *do* loop in Fortran language, etc., of tasks (i.e.,task1,task2) that are completely independent and can be solved simultaneously. In shared address space architectures, the array A is shared by all the tasks. In distributed memory architectures, each task owns a chunk of the array A.

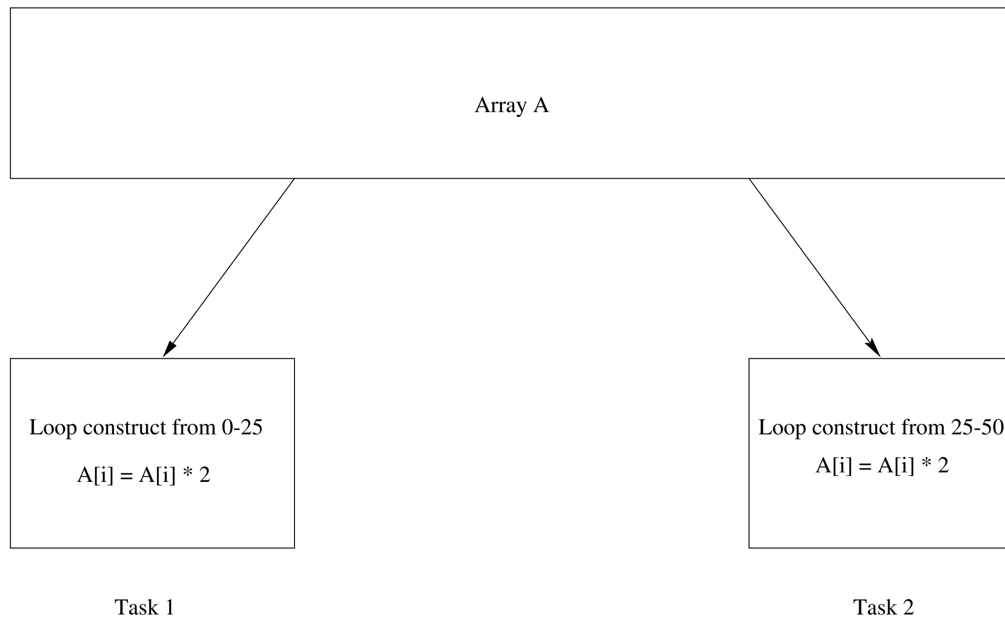


Figure 2.1: Data parallel computation

Loop scheduling refers to ordering of execution of loop iterates onto processors. Proper loop scheduling leads to increased utilization of the processors by reducing the load imbalance among the processors. Scheduling problem can be approached either statically or dynamically. In static schemes, scheduling is performed before runtime whereas in dynamic schemes, scheduling is performed during runtime. Therefore dynamic schemes effectively utilize processors but at the cost of scheduling overhead.

The scheduling problem can be approached as an ordering of tasks with the purpose of minimizing the execution time of a parallel loop with $N$ independent iterates that has to be executed by $P$ processors. A scheduling processor, called the master processor, maintains a work queue of iterates. The master processor selects groups of iterates (chunks) to be assigned to the requesting processor. A processor after finishing its chunk, communicates with the master processor in order to obtain another chunk. When the work queue becomes empty, the scheduler sends an exit message to all the processors in the system.

A generic loop scheduling algorithm can be described as follows.

```
R <- N (total iterations)
repeat
        wait for a request
```

compute the optimal chunk size "k" (where "k" is the number of iterates)

assign "K" to the requesting processor

R <– R - K

until R = 0

where,

R - number of remaining iterations

N - total iterations

K - number of iterates scheduled

There is always uncertainity in calculating the ideal chunk size that would minimize the total execution time. Therefore, all dynamic scheduling techniques are based on probabilistic analysis and they estimate the chunk size such that there is a high probability that they will finish within the optimal time. The following section gives a detailed review of the various dynamic loop scheduling techniques.

### 2.1.1 Static chunking and Self scheduling

These techniques are very simple and straight forward. In static chunking, iterates are assigned to processors in one step. This technique assumes that all the processors are equally faster and the iterates takes same amount of time to execute. Therefore, each processor approximately gets $N/P$ iterates. The optimal case occurs when all the iterates are of constant length and all the processors have the same rate of execution. This technique incurs very little scheduling overhead since the number of scheduling activities is small $(P)$. However, in most cases, this technique may not give optimal performance because of variable rate of execution of processors, and variable length loop iterates.

In self scheduling, each processor obtains a new iterate whenever it becomes idle. All the processors, finish almost at the same time thereby giving effective processor utilization. However, the scheduling overhead may not be acceptable because of large number of scheduling activities $(N)$.

These two techniques can be obtained by setting K as follows in the generic algorithm.

K = N/P for static chunking, and

K = 1 for self scheduling

Although the two naive schemes discussed in this section could not reveal the ideal chunk-size for optimal performance, they suggest that there is a trade-off between allocating iterates in large chunks and in small chunks. Large chunks tend to result in load imbalance, whereas small chunks increase communication overhead.

### 2.1.2 Fixed size chunking

Fixed Size Chunking (FSC) [11] attempts to find an optimal chunk size $k_{opt}$, such that $1 \leq k_{opt} \leq N/P$. For computing $K_{opt}$, it is assumed that *i)* the execution times of iterations are independent identically distributed random variables with mean $\mu$, and standard deviation $\sigma$, *ii)* a constant overhead $h$ is incurred in assigning tasks to a processor, and *iii)* the processors are all initially available at the start of the loop. With these assumptions, the optimal chunk size is found out by using the following approximation

$$K_{opt} = \left( \frac{\sqrt{2}Nh}{\sigma P \sqrt{lnP}} \right)^{2/3} \tag{2.1}$$

In contrast to the static and self scheduling strategies discussed above, FSC takes both the allocation delay and the distribution of the task execution time into account through the ratio $h/\sigma$. However, the optimal chunk size calculated using this formula holds good only when the number of processors is large. Further more, as evident from the formula, the mean execution time $\mu$ is not taken into account. Neglecting $\mu$ however has a significant effect on the chunk size. It is also suggested in [11] that an optimal chunking scheme would use decreasing size chunks instead of fixed size chunks.

An algorithm for fixed sized chunking is obtained by setting K in the generic scheduling algorithm as

$K = K_{opt}$, and

The number of scheduling operation is $N/K_{opt}$

### 2.1.3 Guided Self Scheduling

Guided Self Scheduling (GSS) [12] was proposed to address the problem of uneven processor starting times. This technique is an example of decreasing size chunking strategy. This strategy has the property that unevenness of the execution times of the initial larger chunks will be

smoothed out by the execution times of the later small chunks. The chunk-size is calculated according to the formula, $K = \lceil R/P \rceil$. The number of scheduling operation is $O(P\log(N/P))$. The chunk-sizes calculated according to the formula decrease exponentially. The motivation behind this scheme is that at the time when a chunk is assigned to a processor, all the other processors might be on the point of completing their current chunk. Therefore, it is not wise to give out more than R/P tasks.

This technique assigns the bulk of the iterates initially. This may result in uneven overall finishing times because not enough work in the smaller chunks will be left to smoothen the uneven execution times of the early chunks. This technique will yield optimal finishing time if the iterates have equal running times and if the processors are homogeneous.

### 2.1.4 Factoring

Factoring [3] was proposed for loops with variable iteration execution times with known mean $\mu$ and standard deviation $\sigma$. The iterates are scheduled in batches of $P$ equal sized chunks, and the number of iterates per batch is a fixed ratio of the remaining $R$. The chunk sizes are determined such that they have a high probability of finishing before the optimal time. When a batch is scheduled, a new batch is determined and it is placed at the head of the scheduling queue. The chunk size is determined using the formula,

$$K = \frac{R}{Px_m} \tag{2.2}$$

The optimal values for $x_m$ are computed using the mean $\mu$ and standard deviation $\sigma$ which may not be known before hand. The experiments conducted by the proponents of factoring indicate that $x_m = 2$ works quite well. When there is no variance (no standard deviation) factoring degenerates into static chunking and when the variance is large, factoring degenerates into self scheduling. Guided self scheduling can be viewed as a special case of factoring where each batch contains a single chunk instead of $P$ chunks. Fixed size chunking is also a special case of factoring when there is a single batch of $K_{opt}$ sized chunks.

### 2.1.5 Weighted Factoring

In a heterogeneous environment, processors may have different speed, memory capacity, architecture, and runtime load. During the execution of a parallel loop in such an environment, all these factors have to be taken into account while calculating the chunk size. Weighted factoring [4] is a variant of factoring for scheduling parallel loops on a network of heterogeneous workstations. This technique assigns each processor a constant weight reflecting its relative speed. The weights of the processors are not adjusted after that. Instead of scheduling equal sized chunks within a batch, the strategy allocates a chunk to a processor in proportion to its weight. If $w$ is the weight of the processor, then the chunk size is determined using the formula,

$$K = \frac{w * R}{Px_m} \tag{2.3}$$

### 2.1.6 Adaptive Weighted Factoring

Adaptive weighted factoring [6, 13] evolves from weighted factoring and was designed for time stepping applications. In this technique, the weights of the processors are adjusted at the end of each time step. Initially, all the processors have the same unit weight. Then the weight of a processor is adapted based on its cumulative performance in executing iterates in previous time-steps.

The formulae involved in calculating the weights of the processor is shown below. The Weighted Average Performance (WAP) based on $n$ time steps for each processor $j$ is computed as:

$$WAP_j = \frac{\sum_{(i=1)}^{n} T_{ij} * i}{\sum_{(i=1)}^{n} K_{ij} * i} \tag{2.4}$$

where $T_{ij}$ is the total execution time in processor $j$ to execute $K_{ij}$ iterates in time-step $i$ excluding the time spent on the scheduling procedure.

The average speed of execution AWAP and the reference weight $RWP_j$ of the processor $j$ is:

$$AWAP = \frac{\sum_{(j=1)}^{P} WAP_j}{P} \tag{2.5}$$

$$RWP_j = \frac{AWAP}{WAP_j} \tag{2.6}$$

Equation (2.6) should be normalized with respect to the number of processors P to obtain the actual weight. The sum of all the processors weight is given by:

$$\sum_{(j=1)}^{P} WP_j = P \tag{2.7}$$

Now, the total reference weight is given by

$$TRW = \sum_{(j=1)}^{P} RWP_j \tag{2.8}$$

and the actual weight of the processor is computed by normalizing its reference weight to the total reference weight with respect to all the processors.

$$WP_j = \frac{RWP_j * P}{TRW} \tag{2.9}$$

The chunk-size for the $(i+1)^{th}$ time-step for processor $j$ is given by

$$K_{(i+1)j} = \frac{R_{(i+1)}}{2} * \frac{WP_j}{P} \tag{2.10}$$

### 2.1.7 Adaptive Factoring

Adaptive factoring [5, 14] was proposed for irregular loops in highly unpredictable environments. This technique estimates the rate of change of processor speeds during runtime by collecting information about the iterate execution times. The rate of change of processor speed reflects the nature of the problem, the algorithm and the machine characteristics. The information obtained by capturing the nature of the running environment is incorporated in the selection of the succeeding processor workloads.

The previous methods that were based on factoring inherently assume that the mean and standard deviation of the iteration execution times are known before hand and these do not change during the execution of the loop. However, this is not the case in practical situations. By collecting the execution times of the iterates in a chunk for each processor, this technique

estimates the mean and the standard deviation of the iterate execution time for each processor during runtime and are utilized in the allocation of the succeeding chunks.

The steps involved in calculating the mean and standard deviation are:

**Step 1:** Assign each processor arbitrary $k^{(0)} = N/\ 4P$ iterates and record finishing time of each iterate in every processor. There are $N_1 = N$ - $Pk^{(0)}$ iterates left.

**Step 2:** Estimate mean and standard deviation for each processor based on information obtained from step 1.

$$\hat{\mu_i} = \frac{\sum_1^n X_{ij}}{k^{(0)}} \tag{2.11}$$

$$\hat{\sigma_i} = \left( \frac{\sum_1^n X_i^2 i, j - k^{(0)} \hat{\mu_i^2}}{k^{(0)} - 1} \right)^{\frac{1}{2}} \tag{2.12}$$

where, $X_{i,1}, ...., X_{i,k^{(0)}}$ are the finishing time of iterates in processor $i$ and $n$ is the number of chunks. Now assign

$$k_i^{(1)} = \frac{D_1 + 2T_1 N_1 - \sqrt{D_1^2 + 4D_1 T_1 N_1}}{2\hat{\mu}_i} \tag{2.13}$$

iterates to processor $i$, where

$$D_1 = \sum_{(i=1)}^{P} \frac{\hat{\sigma_i^2}}{\hat{\mu}_i} \tag{2.14}$$

$$T_1 = \left( \sum_{(i=1)}^{P} \frac{1}{\hat{\mu}_i} \right)^{-1} \tag{2.15}$$

Record the finishing time of each iterate in every processor. There are $N_2 = N_1 - \sum_1^P k_i^{(1)}$ iterates left. Assign,

$$k_i^{(n-1)} = \frac{D_{n-1} + 2T_{n-1} N_{n-1} - \sqrt{D_{n-1}^2 + 4D_{n-1} T_{n-1} N_{n-1}}}{2\hat{\mu}_i} \tag{2.16}$$

where $D_{n-1}$ and $T_{n-1}$ are computed by the (2.14) and (2.15) respectively but use the new estimator of the means and standard deviation.

### 2.1.8 Fractiling

Fractiling [15] was proposed for load balancing N-body simulations. It uses the same formula as factoring to compute the optimal batch chunk size.

Fractiling is a combination of both factoring and tiling strategies. Tiling statically partitions the iteration space into tiles whose shape is carefully chosen to maximize data reuse and locality. During dynamic load balancing, re-assignment of work may need access to remote data and careful placement of data on processors can significantly reduce the data access latency. In fractiling, the decreasing size chunks are represented by sub-tiles called fractiles of same shape. This maximizes data reuse. Fractiling simultaneously balances processor loads and maintains data locality by exploiting the self-similarity properties of fractals.

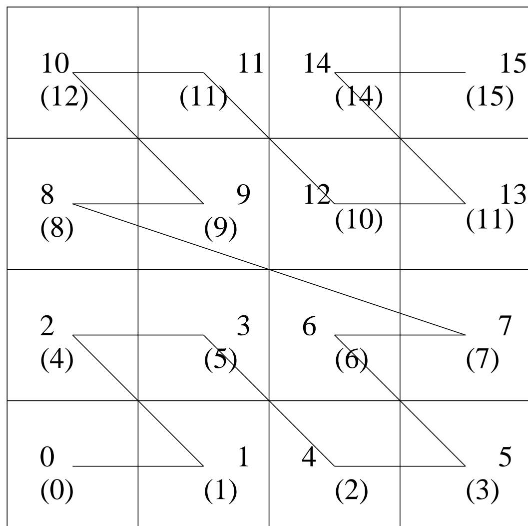


Figure 2.2: Morton ordering

Morton ordering helps to preserve the self-similarity properties of fractals and to exploit its locality. Figure 2.2 is an example of two-dimensional Morton ordering. In order to effectively partition computations, a self-similarity tile allocation scheme is necessary. Self-similarity property allows the sub-tiles to have the same shape of the whole tile. In general, it is not possible to divide a d-dimensional shape into two similarly shaped parts. However, it is possible to divide a d-dimensional shape into $2^d$ similarly shaped parts by bisecting each dimension. The self-similarity properties of the fractals allows the easy maintenance of the execution order as shown in the Figure 2.2.

All of the scheduling techniques previously discussed (except Adaptive Factoring and Adaptive Weighted Factoring) accommodates only algorithmic variance whereas fractiling accommodates both algorithmic and systemic variance. This gives fractiling an advantage over those scheduling techniques.

Table 2.1: Chunk sizes for different scheduling schemes

| Scheme | N=800, P=4, Weights=1.5,0.5,1,1 |
|---|---|
| static chunking | 200,200,200,200 |
| Self scheduling | 1,1,1,1,..... |
| Fixed-size chunking | K,K,K,K,.... |
| Guided self scheduling | 200,150,113,85,.... |
| Factoring | 100,100,100,100,50,50,50,50,.... |
| Weighted Factoring | 150,50,100,100,75,25,50,50,.... |

## 2.2 Runtime System

A runtime system is a library of software primitives that are designed to efficiently handle complex, irregular scientific problems. They support features like process creation, management, synchronization, destruction, etc. By shielding the users from the low-level machine specific issues, runtime system helps in easy implementation of computational problems on parallel machines. Thus, runtime system have a significant impact on the performance and portability of the scientific applications. This section presents an high level overview of some of the commonly used runtime systems.

### 2.2.1 Nexus

Nexus [16] is a runtime system and is currently being used as a compiler target for task parallel languages like Fortran M and Compositional C++. In Nexus, the computation consists of a set of threads and each thread executes in an address space called the context. The contexts can be dynamically created and destroyed. One or more contexts can be mapped onto a node (processor).Nexus also provides the compiler, the concept of a global pointer. Nexus supports the remote service request (RSR) which is similar to active messages [17]. Using remote service request, a thread can request an action be performed in a remote context. A remote service

request results in the execution of a special function called the handler. The handler is a new thread that is invoked upon receipt of an incoming message.

### 2.2.2 Tulip

Tulip [7] is a runtime system designed specifically to support object-parallelism. Basic communication supported by Tulip are through global pointers, remote load and store and RSR driven communication. Unlike Nexus, Tulip supports collective communication primitives as well. Tulip is used as a parallel runtime system by pC++ parallel programming language. The object oriented style of Tulip makes itself more amenable to a single-sided communication model. Unlike Nexus, Tulip can also be used as an API by library designers.

### 2.2.3 C Region Library

C Region Library (CRL) [8] is an all-software distributed shared memory (DSM) system. CRL is a lightweight, portable library and do not rely on special language, operating system or hardware support. Applications that use CRL share data through regions. A region is any contiguous area of memory. Regions can be dynamically created and destroyed. CRL allows access of a region only after the processor maps the region on to the local address space. CRL handles all communication through active messages. CRL also provides global synchronization operations like barrier, broadcast, reduction, etc. Like most hardware DSM systems, CRL employs a fixed-home, directory-based invalidate protocol for cache-coherence.

### 2.2.4 TreadMarks

TreadMarks [18] is another DSM system that provides shared memory abstraction to an application on a network of workstations. TreadMarks provides facilities for process creation, process destruction, synchronization and shared memory allocation. TreadMarks uses the lazy release consistency algorithm [19] to implement release consistency.

### 2.2.5 MOL/DMCS

The mobile object layer (MOL) and data movement and control substrate (DMCS) [9, 10] is a portable runtime system developed for implementing parallel adaptive applications. The

MOL/DMCS system is a two-layered system. The MOL is constructed on top of DMCS. The DMCS provides a one-sided communication API similar to Nexus and Tulip. The top layer, the MOL, supports global addressing scheme in the context of object mobility as provided by systems like CRL and TreadMarks. The MOL also supports object mobility like Emerald [20] and Amber [21]. But Emerald and Amber are full-fledged, high-level, object-oriented languages. The MOL forwards any misdirected message to the right node through automatic message forwarding mechanism.

### 2.3 Parallelizing C Compiler

The Sun ANSI/ISO C compiler can generate parallel code that can run on shared-memory multiprocessor machines. The C compiler parallelizes those loop structures that are safe to parallelize. The user can also give explicit commands to parallelize any loop structure. Also, the compiler includes a number of dynamic loop scheduling techniques like self scheduling and guided self scheduling.

### 2.4 Static partitioning

In many scientific applications, graph partitioning is one of the fundamental problems. Some of the commonly used graph partitioning software packages include Chaco [22] and Metis [23]. These software packages comprise of partitioning algorithms like inertial, spectral and multilevel methods. Since the decomposition is carried only once, these algorithms are called static partitioning algorithms. These packages use a file-based interface i.e. they read geometrical information about the graphs from a file, partition them and again writes back to the file which is then read by the application. Therefore, static decomposition can be carried out as a pre-processing step and are often done sequentially.

### 2.5 Dynamic partitioning

Zoltan [24] is a general purpose dynamic load balancing tool that contains a suite of load balancing algorithms for doing both geometry based and graph based decomposition. The suite of algorithms included in Zoltan are recursive co-ordinate bisection, recursive inertial bisection,

Hilbert space-filling curve partitioning, etc. These algorithms are called dynamic algorithms because the decomposition is adapted as the underlying mesh changes. Therefore, decomposition becomes a part of the problem itself and cannot be carried out entirely as a pre-processing step. The algorithm must run in parallel and should be fast enough to minimize the cost of new decomposition.

## 2.6 Summary

Significant amount of work has been done in the area of runtime systems. Developing parallel applications is a time-consuming and error-prone task, especially without the aid of software tools. A runtime system is a library of software primitives that efficiently handles the complex requirements of irregular scientific problems. A number of runtime systems supports features like one-sided communication primitives, remote service requests, etc. However, there is no runtime system that supports features like global pointers and object mobility like MOL/DMCS. Though, Emerald and Amber support these features, they are full-fledged, high-level, object-oriented languages. In the area of loop scheduling, a great deal of work has been done that has resulted in advanced loop scheduling techniques like adaptive factoring, adaptive weighted factoring, etc. But there is no single technique that can effectively satisfy the needs of all the scientific applications. Different techniques work well for different kinds of application. Therefore, there is a need for integrating all the loop scheduling techniques into a single package and provide them as a common interface to the application developer. Along this direction, the sun ANSI/ISO C compiler has integrated few dynamic loop scheduling techniques like self-scheduling and guided self scheduling for shared-memory multiprocessor machines. Moreover, most of the currently available static and dynamic load balancing tools discussed in the chapter use iterative static approach that do not seem to address irregularities that occur during the execution, for instance within a time step of the computation [6]. So, there is a need for load balancing techniques that can address all the sources of load imbalance.

To the best of our knowledge, a combined approach of integrating dynamic loop scheduling techniques into a runtime system (MOL/DMCS) to address the performance degradation of parallel applications due to load imbalance for distributed memory machines, has not been

pursued yet. Such a combined approach would expand the applicability of the MOL/DMCS system.

# CHAPTER III

# LIBRARY DESIGN

Having laid out some of the underlying principles of the related areas of research in the previous chapter, the main goal of this chapter is to describe the design details of the dynamic load balancing library (DLBL) - the library package which we intend to build. This chapter is organized as follows. A brief discussion about the various communication modes is given in section 3.1. The underlying principles of the runtime system data movement and control substrate (DMCS), and mobile object layer (MOL) are outlined in section 3.2. In section 3.3, an overview of the DLBL library is presented. Different memory allocation strategies are discussed in section 3.4. In section 3.5, a description of the handlers used by the DLBL library is given. In section 3.6 and 3.7, a detailed explanation of the master-slave strategy for two-sided and one-sided communication model is given.

## 3.1 One-sided vs Two-sided communication

In distributed memory machines, communication (message passing) between the processors is established using message passing libraries like MPI (Message Passing Interface) or PVM (Parallel Virtual Machine). Both these libraries are two-sided models meaning that both the sender and the receiver have to explicitly take part in the communication i.e., communication and synchronization are coupled together. These libraries provide both synchronous as well as asynchronous communication calls. In asynchronous calls, the computation on the processor that sends message (sender) can resume as soon as the message is on its way to the destination processor. However, during synchronous calls, the computation on the sender cannot proceed until the matching receive is posted by the receiving processor. These two-sided models are acceptable communication model for bulk transfer of data.

In a one-sided communication model, only the sender is responsible for initiating the communication between two processors. The sender alone would supply all the parameters needed for the communication operation. The communication model which is more suitable for a given problem is determined by the algorithmic structure of the application [25]. For applications with unstructured communication patterns, two-sided models may not be appropriate for the reason that they would increase the complexity of the code in order to handle the uncertainity in placing the receive calls. Because of the nature of two-sided communication model, receive calls are necessary to gather messages that may or may not have been sent. On the other hand, a one-sided communication model with asynchronous remote procedure calls would greatly simplify the logic of the code and would significantly improve the performance of the code.

### 3.2 Parallel Runtime Environment

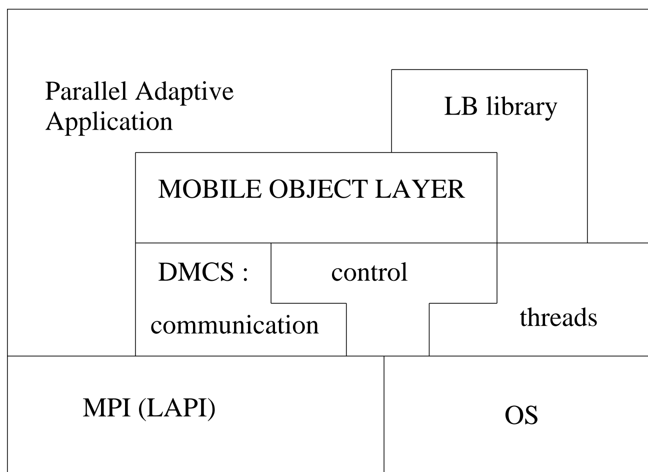


Figure 3.1: Parallel runtime system architecture

The DLBL system is built on top of a runtime system called parallel runtime environment for multicomputer architecture (PREMA). The PREMA is a lean, language-independent and easy to port and maintain runtime system for implementing adaptive applications on current and non-traditional parallel platforms [9]. The design philosophy behind PREMA is the principle of *separation of concerns* [9]. Figure 3.1 [10] represents the overall system architecture of PREMA and the layers which address the different requirement of parallel applications.

#### 3.2.1 Data Movement and Control Substrate

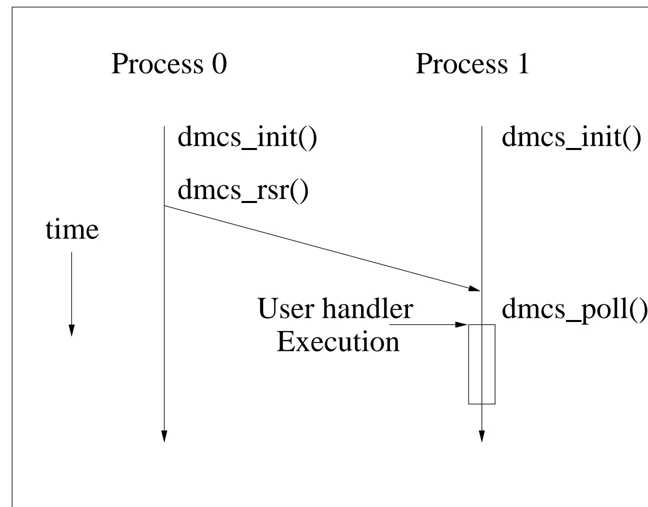


Figure 3.2: DMCS Execution Model

The execution model of DMCS is shown in Figure 3.2 [10]. The DMCS provides one-sided, low latency communication primitives for parallel adaptive and irregular applications [10]. The DMCS is built on top of low level messaging layer like MPI on distributed memory machines and low-level application programming interface (LAPI) for IBM SP machines. The DMCS provides flexible one-sided communication primitives like put, get and remote procedure calls [10]. The DMCS is a single threaded execution model which means there can be only one application thread during program execution. One advantage of having a single threaded execution model is that the computation and communication are not interrupted. The user handlers are executed only during the polling phase.

The dmcs_init() call initializes the DMCS environment which is similar to MPI_Init(). The dmcs_rsr() is a remote service request call. The DMCS provides both the synchronous as well as the asynchronous version of this call. All remote node operations like dmcs_get(), dmcs_put(), and dmcs_rsr() are executed during the polling phase which is indicated by dmcs_poll().

#### 3.2.2 Mobile Object Layer

Many parallel adaptive applications are characterized by dynamic, data dependent, irregular computation and communication requirements. These characteristics exacerbate the

programming complexities of developing an efficient parallel code. This inherent complexity necessitates the use of software tools and libraries. The mobile object layer (MOL) [9] is a software system that eases the burden on the application developer by providing the building blocks for parallel adaptive applications.

As illustrated in Figure 3.1, MOL is constructed on top of DMCS. The MOL supports object mobility and automatic message forwarding which eases the implementation of parallel and adaptive applications. The MOL provides mechanisms to create mobile objects. The mobile objects are held together by mobile pointers. Once the mobile objects are built, the objects may be moved from one processor to another processor and all the mobile pointers will still remain valid. The MOL keeps track of all the mobile objects. Internally each processor maintains a lookup directory which gives the location of the mobile objects. When a processor sends a message to the mobile object, the message is sent to the location shown in that processor's directory. If that location happens to be incorrect, the MOL forwards the message to the right location and sends an update back to the processor that sent the message. This forwarding of message to the right processor is known as *automatic message forwarding* mechanism. The structure of a mobile object is dictated by the application, and the application must explicitly instantiate the mobile object. Also, the application must specify when and where to move the mobile objects. This concern is addressed by the load balancing library.

## 3.3 The DLBL library

The DLBL is developed with the goal of providing the application developers a general purpose dynamic load balancing tool that can be easily used by data parallel scientific applications. The library provides routines specifically for executing parallel loops on distributed memory systems. The library implements several dynamic loop scheduling algorithms and is designed in such a way that new algorithms can be easily added. The library provides the following functionalities.

1. Data migration : This consists of migrating data from one processor to another processor during runtime due to load balancing.

2. Scheduling : The scheduling algorithm determines how many loop iterates are to be executed by a processor and when data should be moved from source processor to a target processor.

### 3.3.1 DLBL structure

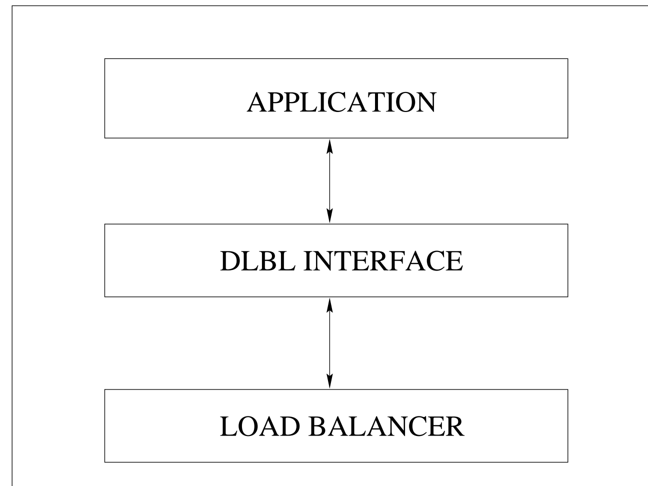


Figure 3.3: DLBL architecture

Figure 3.3 depicts the view of DLBL from operating system's point of view. For the operating system, DLBL is a part of the application program because the load balancing is done at the application level. From the application's point of view, DLBL is simply a library which is linked to the application.

### 3.3.2 Phases of load balancing

The load balancing problem can be split up into two phases as follows.

1. Decision phase : Initially, the input data is equally divided among the available processors. The decision phase starts when a processor finishes the work corresponding to its share of the data and requests for more work. In this phase, the load balancer decides whether data has to be moved and if so, determines how much data has to be moved and identifies the processor that will be the source of the data (sender).

2. Load migration phase : In this phase, the actual data migration takes place. After identifying the sender in the decision phase, the load balancer will send a message to the sender to yield work.

## 3.4 Memory Allocation Strategies

The DLBL is basically a scheduling tool that decides how much data has to be moved across processors during load balancing phase. However, DLBL does not make any assumptions about the application's data structure. It is entirely up to the user to build the data structure and allocate space for data movement during load balancing. Two strategies can be employed for memory allocation during data redistribution. They are:

### 3.4.1 Strategy I

Consider the following sequential "for" loop construct, where each loop iterate

```
for (i=0; i<512; i++)
    doWork(i);
```

modifies an element in an array of structures. In a uniprocessor environment, since only a single CPU does all the work, all the data will be owned by a single processor. However, the scenario is totally different when we solve the problem in parallel. The ideal scenario (with less memory utilization) will be as follows.

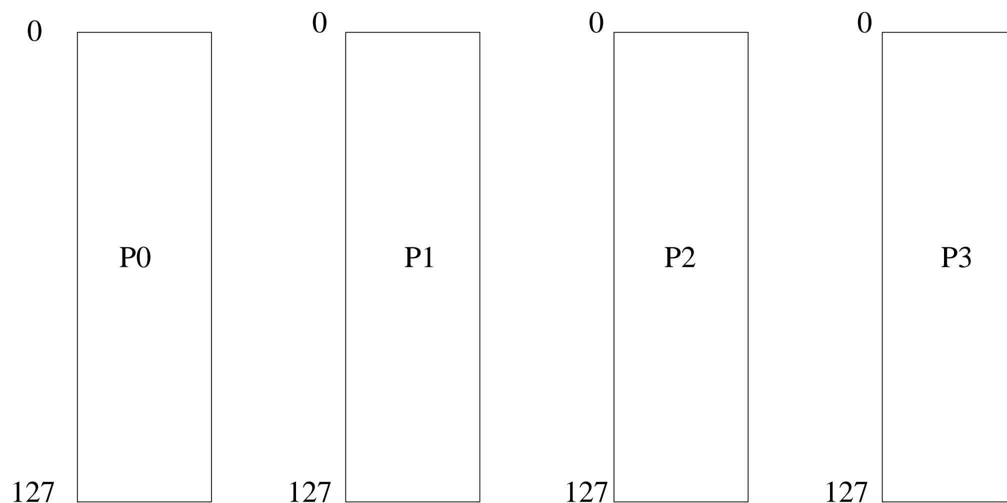


Figure 3.4: Strategy I

In this strategy, each processor will own only its share of data. In the given example, total loop iterates are N=512, and number of processors is P=4. Therefore, the share of each processor is n=N/P=128 iterates. For a straight forward parallelization, in each processor, memory will

be allocated only for N/P data items. In each processor's memory, a local-index will start from 0 and run to n=N/P. There is another index called "global-index", which for the example under consideration, runs from 0 to 511. It is important to understand the relationship between global-index and local-index because DLBL uses both indices. The relationship between the local-index and global-index is as follows:

$$(local - index) + n * (proc - id) = (global - index) \tag{3.1}$$

For example, the local-index of 5 owned by processor 1 corresponds to 5 + 128*1 = $133^{rd}$ global-index.

The Figure 3.4 illustrates the data distribution among the available processors for straight forward parallelization i.e. without any load balancing. During runtime, load balancing phase may necessitate data redistribution from one processor to another processor. A typical scenario is shown in Figure 3.5. Suppose data corresponding to local-index 50 through 75 are to be moved from processor 1 to processor 0. When data arrives at processor 0, a temporary buffer must be dynamically allocated for it before processor 0 can work on the data.

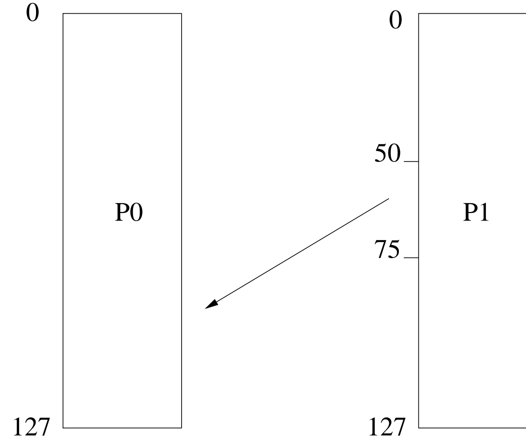


Figure 3.5: Data movement

Large, memory intensive applications would be programmed in this way. The DLBL will decide how much data will be moved from one processor to another processor. However, it is up to the programmer to allocate buffers for storing any incoming data.

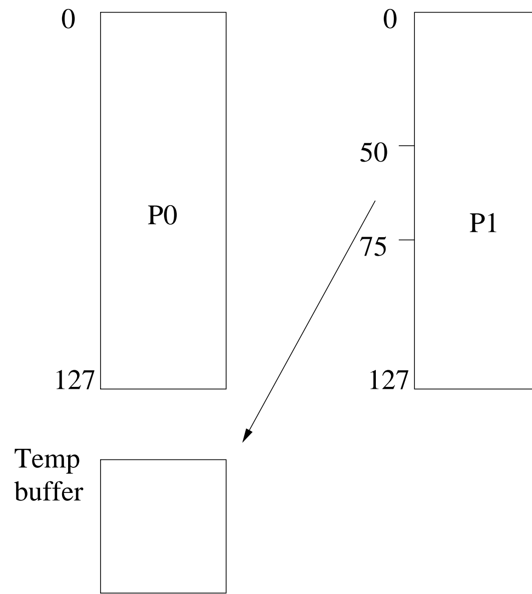


Figure 3.6: Buffer allocation

### 3.4.2 Strategy II

If dynamic allocation of temporary buffers for storing migrated data is not desirable, memory can be allocated in a single step that can accommodate any data transfer during runtime. In such a scenario, buffers for holding data would look like Figure 3.7. The maximum amount of buffer space needed by any processor is limited by the total loop iterates (in this case, it is

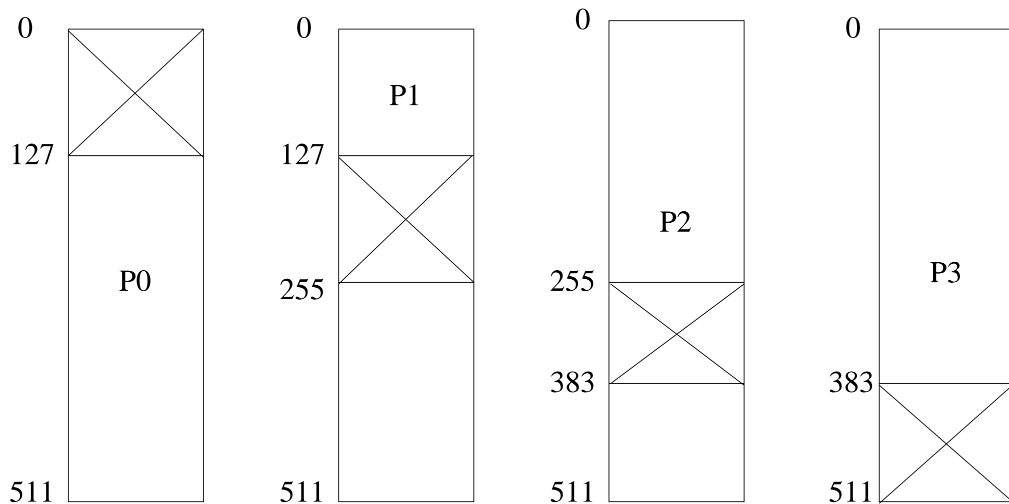


Figure 3.7: Strategy II

512). If possible, a buffer that can accommodate any incoming buffer can be allocated on each processor's memory. This strategy has a clear disadvantage of not utilizing all the allocated space. Therefore, this strategy cannot avoid some wastage of memory. The only advantage is the dynamic allocation and deallocation of buffer space for any incoming data is eliminated. This strategy uses global indexing mode.

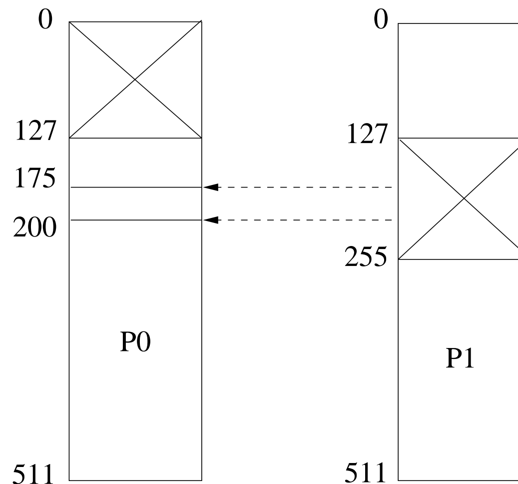


Figure 3.8: Data movement and storage

The Figure 3.8 illustrates data redistribution. During data migration, data that arrives on any processor is stored in global-indexing mode i.e. if data is migrated from global index 150 to 200 from processor 1 to processor 0, they are stored at the same locations in processor 0.

Since it is possible to have different data ordering, it would be naive to develop a library for each type of ordering. The DLBL library is built according to strategy II. The user can always go from one index to another index according to the equation (3.1)

### 3.5 Handlers

A handler is a user-defined function that is executed during a polling operation. The PREMA system operates in a polling mode. This means any incoming message will not interrupt the computation but instead will be queued for execution during application posted polling operation. The DLBL system requires the definition of five different handlers for different purposes. They are:

1. own_work_handler (The work routine to be load-balanced).

2. others_work_handler (The work routine to be load-balanced).

3. yield_work_handler (Routine to pack the migrated data).

4. unpack_work_handler (Routine to unpack the migrated data).

5. exit_work_handler (Routine to exit a processor).

These handlers are required by the underlying runtime system PREMA, and are invoked inside the DLBL library. The user need not explicitly invoke any of them. The handlers are defined inside the DLBL system.

It is impossible for a single library to efficiently satisfy the needs of a broad range of irregular, parallel applications. So, the user must perform some amount of work in order to load balance his/her application. The following responsibilities lies with the user.

1. Computational routine : the doWork() routine.

2. Routine for packing the data (Not required for version II) : This routine is executed by a processor identified by the load balancer as a source of work.

3. Routine for unpacking the data : executed by a processor that is the recipient of additional work.

The general form of the handler as required by the runtime system PREMA is as follows.

*void handler_name(int src, void *data, int size, void *arg)*

where

1. src is the processor-id from which the handler was executed.

2. data is any data that may be passed from one processor to another processor.

3. size is the size in bytes of the data sent.

4. arg is any other argument that may be passed to the handler.

These handlers are called *remote handlers* and are similar to remote procedure call (RPC) except that they cannot return any value [10]. Typically, the user supplied routines like work routine, pack data routine, unpack data routine will be invoked inside one of these five handlers.

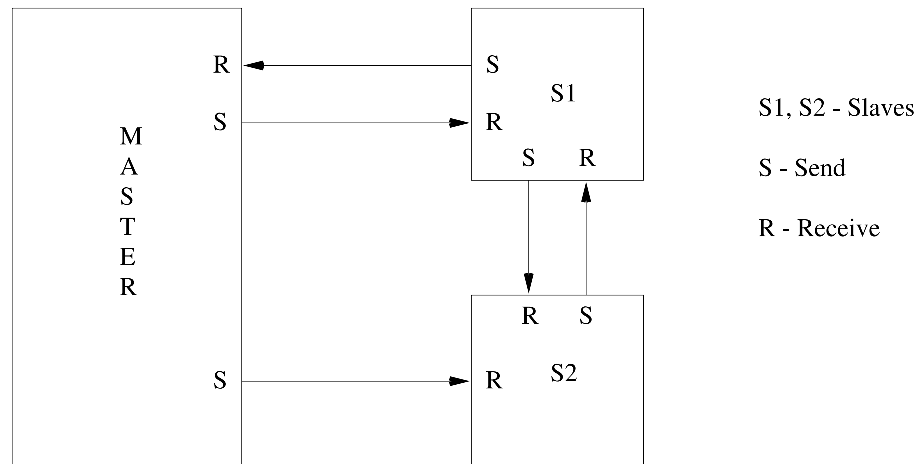


Figure 3.9: Master-Slave strategy for two-sided model

### 3.6 Master-Slave Strategy for Two-Sided Model

In a traditional master-slave strategy, tasks are scheduled by a processor usually referred to as the master and the slave processors perform the useful computations required to solve the problem. The master processor executes the load balancing algorithm in addition to performing the work of a slave. The master processor performs the load balancing task only upon receipt of a request from a slave processor. The Figure 3.9 illustrates the master-slave strategy for a two-sided communication model.

In the master-slave strategy, to start with, the input data is equally divided among the available processors. The slaves are allowed to work on their own data only after requesting the master processor. The master processor maintains a table called the *chunk-table* for load balancing purpose. The *chunk-table* contains information about the amount of work scheduled on each processor. After receiving the request from a slave processor, the master processor, based on the corresponding scheduling algorithm, determines how much to schedule on the requesting processor. The master processor will then schedule the minimum of the amount of work left in the requesting processor's own data space and the current amount of scheduled work. In any case, the master processor will send the start and the end index to the requesting processor. The slave processor after receiving the indices, will start working on that portion of the data.

Consider a scenario in which a slave processor has exhausted its portion of the work. As before, the slave processor will request the master processor for work. The master processor

after receiving the request will determine the amount of work to be scheduled on the requesting processor. By looking at the *chunk-table* the master processor will understand the requesting slave processor has finished its portion of work and now it is ready to help any slow processor in the system. Again, by looking into the *chunk-table*, the master processor can identify the slowest processor in the system. Now, the master processor have to send two messages: one to the requesting processor and one to the slowest processor. The slowest processor after receiving the message from the master processor, will yield the data to the helping processor. The requesting processor, after receiving work from the slowest processor will work on that data and send the results back to the source (slowest) processor. Once all the scheduling activities are over, the master will send an exit message to all the slave processors.

In the above strategy, there is an explicit synchronization between the slaves and the master processor for all kinds of messages. Moreover, every message has to be received with a different tag. Depending upon the tag, appropriate action would be taken. Thus, one of the consequences with two-sided communication models like MPI-I is the increase in code complexity. The next section explains the Master-Slave strategy for one-sided communication model.

### 3.7 Master-Slave Strategy for One-Sided Model

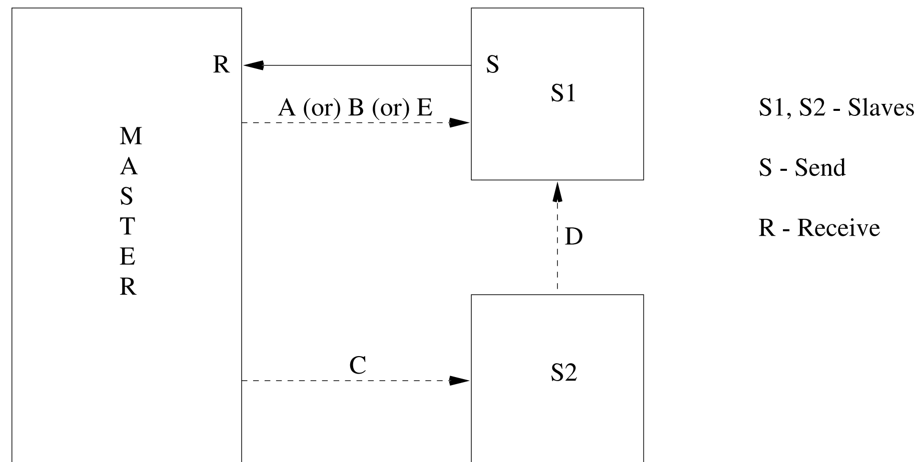


Figure 3.10: Master-Slave strategy for one-sided model

The working semantics of the master-slave strategy for a one-sided communication model is same as that of the two-sided communication model except the way in which communications

are handled. The Figure 3.10 represents the master-slave strategy for a two-sided communication model. The alphabets in the figure represents the following handlers.

1. A : own_work_handler
2. B : others_work_handler
3. C : yield_work_handler
4. D : unpack_work_handler
5. E : exit_work_handler

As mentioned before, the DLBL system handles everything in the form of handlers which are executed during the polling phase. In this communication model, the slave processors will go to polling mode immediately after requesting the master processor for work. Usually one of the following three handlers will be executed in the polling phase: the own_work_handler, others_work_handler, and the exit_work_handler.

The scheduling process starts when one of the slave processors request master processor for work. The master processor, upon looking into the *chunk-table* determines what handler to execute on the requesting processor. When the requesting processor has work left on its own data space, the master processor will execute the own_work_handler on the requesting processor. When the requesting processor has completely exhausted its portion of work, the master identifies the requesting processor as the helping processor. By looking into the *chunk-table*, the master processor determines the slowest processor. Now, the master processor executes others_work_handler on the helping processor and yield_work_handler on the slowest processor. When the master processor executes the yield_work_handler on the slowest processor, the slowest processor will be busy executing the work routine invoked by the previous own_work_handler from the master processor. The yield_work_handler will be executed in the polling phase which is inside the own_work_handler. In order to respond to the yield_work_handler while busy with the computations, the work routine must invoke the DLBL library routine *dlbl_check()* from time to time. If *dlbl_check()* confirms that a yield_work_handler was executed, the slowest processor immediately suspends the computations, move data to the processor specified by the handler and then resume the computations.

Since the DLBL system is not aware of the data structure of the application, the user must provides routines to pack the data to be migrated and unpack the migrated data. The DLBL system takes care of the packed data by migrating it to the destination processor. When the helping processor executes the others_work_handler, it involves a polling phase. The unpack_work_handler is executed inside this polling phase. The entire sequence of data movement happens like a chain of events. First, the master processor executes the yield_work_handler on the slowest processor. When the slowest processor handles this request during the polling phase, it will trigger the execution of the unpack_work_handler on the helping processor. When the helping processor handles this request during the polling phase, the migrated data will be unpacked and stored into some local buffer and the helping processor will start working on that data. The last phase of the action i.e. storing the passed data and working on it happens during the execution of the others_work_handler.

The helping processor does not immediately send back the results of the computations to the original processor as shown in Figure 3.9. This is because, doing so may lead to network contention and may delay load balancing messages from the master processor. So, this message was uncoupled from all other scheduling messages, which means the responsibility of sending back the results to the original processor lies with the user. However, the user need not keep track of which data moves from where to where. The DLBL system keeps track of the locations of all the results for the user. Once all the scheduling activities are over, the results can be sent back to the original processors. This strategy also has another advantage. Instead of sending many small messages, the small messages can be combined into a single large message. This reduces the communication latency penalty every-time a message is sent. Networks characterized by large communication latency can obtain significant performance improvement by this method. Once all the scheduling activities are over, the master processor executes the exit_work_handler on the requesting slave. The slave processors exits from the polling phase only after the exit_work_handler is executed by the master.

The version II of DLBL library makes use of mobile objects and mobile pointers supported by MOL. The mobile objects are objects of application's data structure and mobile pointers points to mobile objects. The master-slave strategy for this version is similar to the one showed in Figure 3.10 except the user need not provide the routine for packing data during data redistribution. The application programmer is required to create mobile objects and mobile pointers using the

mechanism provided by PREMA. Once the mobile objects and pointers are created, they are passed to the DLBL system. The DLBL system migrates the mobile objects and pointers as and when needed during the runtime. As before, the user is required to provide a routine to unpack the objects that arrive at the helping processor.

Finally, the DLBL system augments the underlying runtime system PREMA and the low level messaging layer MPI without obscuring access to it. The application developer still has complete and direct access to the underlying communication substrate.

### 3.8 Summary

The intent of this chapter was to describe the design details of the load balancing library. We started the discussion by describing the two-sided and the one-sided communication model. We saw that both are acceptable model and choice of any particular model depends upon the algorithmic structure of the application. However, adaptive applications with unstructured communication pattern requires one-sided communication model. We then saw a brief description of the runtime system PREMA. The runtime system supports features like one-sided communication, remote service requests, and object mobility. The functionalities provided by DLBL are scheduling and data redistribution. During data movement, there are two ways to allocate memory for storing any incoming data : (i) storing in a preallocated buffer and (ii) allocating and deallocating buffer as and when needed. We saw that advantages and disadvantages of both the strategies. We saw that handlers form an integral part of the load balancing library. There are five handlers for five different purposes. The users are not required to define these handlers. However, what goes into the handlers are to be defined by the users. Finally, we saw the master-slave coordination for two-sided and one-sided communication model. The problem with the two-sided model are the increase in the code complexity and the coupling of communication and synchronization operation.

With the integration of the scheduling techniques into the runtime system, we have augmented its features. The performance analysis of the techniques and their overhead are presented in the subsequent chapters.

# CHAPTER IV
# LIBRARY EVALUATION

In this chapter we describe the test applications and the performance metrics used in order to assess the applicability of the scheduling techniques. The results obtained are then interpreted. This chapter is mainly divided into three parts. In the first part of the chapter, a description about the different modes of evaluation of the developed library is given. The experimental setup and the performance metrics are discussed in the second part of the chapter. In the final part, the test applications are described followed by the interpretation of the results.

## 4.1 Modes of Evaluation

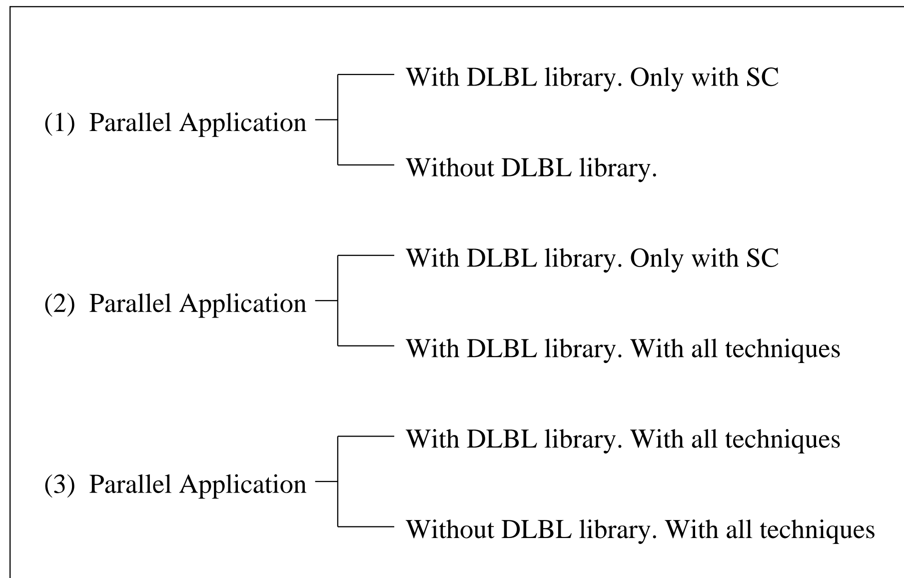


Figure 4.1: Different modes of evaluation

The Figure 4.1 depicts the different kinds of evaluation used to analyze the library. The numbers in the Figure 4.1 indicates the type of evaluation being made: (1) This evaluation is

made between the parallel application without the library and the parallel application with the library using static chunking. This comparison will give an idea of the overhead suffered by the application just by linking to the dynamic load balancing library (DLBL) library. We expect the overhead to be very small. (2) This comparison is between the parallel application without load balancing and the parallel application with the library using different scheduling techniques. We expect the benefits of load balancing to outweigh the overhead introduced by scheduling operations. (3) This comparison is between the DLBL library and another load balancing library which entirely uses native message passing layer. We expect this comparison to give an idea of the overhead suffered by the PREMA application.

The first two evaluations are presented in this chapter and the third evaluation is presented in the next chapter (chapter V).

## 4.2 Library overhead

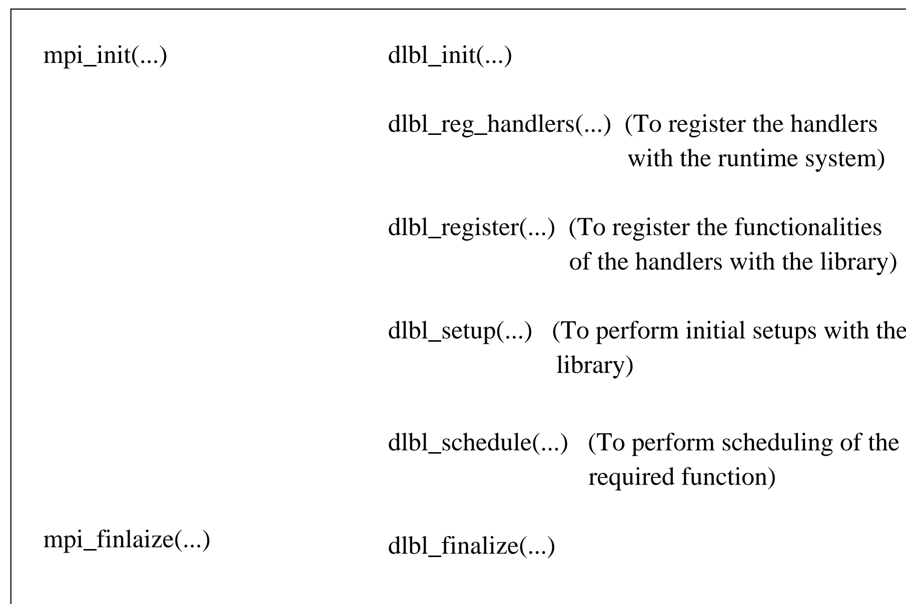


Figure 4.2: Sample DLBL application

The library overhead is evaluated by running the parallel application without the library and the parallel application with the library using the straight forward parallelization technique. Since, all forms of overhead results in increased cost, this comparison is made by comparing

Table 4.1: Parallel application cost with and without the library

| Application | Input file | Cost without library (secs) | Cost with library (secs) | % cost increase |
|---|---|---|---|---|
| Nbody | u100k | 1268 | 1486 | 17.19 |
| | c100k | 3632 | 3836 | 05.62 |
| | g100k | 14178 | 14452 | 01.93 |
| AQR | 15120f | 22347 | 22618 | 01.21 |
| | 15120b | 22680 | 22950 | 01.19 |
| | 15120c | 22763 | 22832 | 00.30 |
| | 15120s | 20747 | 21454 | 03.41 |

the respective cost. Since, the library is tested exhaustively for different number of problem sizes and different number of processors, it would be naive to compare the results for all those testings. Therefore, the results are shown only for the largest problem sizes for all the test cases considered. The cost metrics shown in table 4.1 are for the largest number of processors on which the particular test case was run. The Figure 4.2 shows the sample DLBL application.

The table 4.1 represents the cost of running the application with and without linking to the DLBL library. A detailed description about the application and the input files are given in the following sections. For now, in the $N$-body application, u100k represents the uniform distribution for 100k particles, c100k represents the corner distribution for 100k particles, and g100k represents the Gaussian distribution for 100k particles. For the AQR application, 15120 represents the number of integrals to be evaluated, and f,b,c,s represents the front, back, center, scatter distribution respectively. While comparing the columns $3^{rd}$ and $4^{th}$ in table 4.1, it can be noticed that the values in the column 4 are always found to be comparable with the values in the column 3 except the first row. This observation can be attributed to the fact that there is not much load imbalance in the uniform distribution. However, with increase in the load imbalance of the problem, we see the % cost increase is less than 6%. This shows that there is very little overhead in just linking the library with the application.

## 4.3 Experimental setup

All experiments were conducted on SUN ultraMSPARC cluster at the Engineering Research Center (ERC) at the Mississippi State University. The ultraMSPARC is a 16 node, 64 processor cluster. All processors are of ultraSPARC II architecture with 400 MHz of processing speed. The cluster has a combined RAM capacity of 32 GB. The nodes are interconnected through a Myrinet switch as well as 100 Mb/s Ethernet switch.

## 4.4 Performance Metrics

The work (N) is defined as the minimum number of operations required to complete a given task. The execution time ($t_p$) represents the time required to complete a given task. The execution time depends on a number of parameters like the algorithm used, the underlying architecture, the processor speed etc. The performance can be defined as the ratio of the work to the execution time. Three performance metrics are defined in the following subsections.

### 4.4.1 Cost

Cost is a measure of resource utilization. In a scalable parallel system, when $p$ processors are utilized, the cost is the product of the number of processors used and the execution time. Cost is a direct indicator of the performance of the algorithm. The lower the cost, better the algorithm.

$$\text{Cost} = p\text{X}t_p.$$

### 4.4.2 Effectiveness

This metric was proposed by [26]. Denoted by $\Gamma$, this metric can be defined as the ratio of performance to cost. High effectiveness value indicates the better performance of the algorithm.

$$\Gamma = \frac{N}{pXt_p2}$$

### 4.4.3 Coefficient of Variation

This metric measures the variance of the processors finishing time. More variation in the processors finishing time implies that the work is not properly distributed among the processors.

Together with the cost performance metric, this metric gives an idea of how effectively the processors were utilized in solving a given task. Let $x_i$ be the execution times of the individual processors, $\mu$ be the mean of $x_i$ and n be the number of processors. Then,

$$\sigma = \sqrt{\frac{\sum_{(i=1)}^{n}(x_i-\mu)^2}{n-1}}$$

$$c.o.v = \frac{\sigma}{\mu}$$

### 4.5 $N$-body Simulations

Given the positions and velocities of "n" particles, the problem is to simulate the evolution of particles over time. The $N$-body simulations find its application in astrophysics, molecular dynamics, plasma physics, etc. The simulation proceeds over a number of time-steps, and in each step, the net force acting on each particle is computed. Every particle in the system experience force due to its interaction with every other particle in the system. Therefore, the naive algorithm, which is based on inverse square law, is of $O(N^2)$ complexity. With the scaling up of number of particles in the system, the naive algorithm becomes practically infeasible because of its complexity. Therefore, several approximation algorithms were proposed. Some of them include: O(NlogN) [27], O(N) [28], and O(N) [29]. This implementation uses Greengard's Fast Multipole Algorithm (FMA) which has O(N) complexity.

The FMA algorithm use a tree structure to represent the hierarchical decomposition of the physical space. The algorithm recursively partitions the physical space into a hierarchy of finer grained cells, resulting in either a quad-tree (2-dimensions) or an oct-tree (3-dimensions). The FMA algorithm also maintains control over the accuracy of the computation. Multipole expansions, which are commonly used in problems like gravitational field of mass aggregations, the electric and magnetic fields of charge and current distributions, is used. The lowest level of the tree which corresponds to the leaves of the tree, consists of particles. The algorithm requires two traversals over the tree. In the upward pass of the tree, the summary of the field effects of particles in the subtrees are propagated up the tree and in the downward pass of the tree, the local expansions and the direct interactions are computed. The interactions of the nearby particles are calculated directly and the interactions between particles that are sufficiently far away are computed using multipole expansions. The net multipole effect is represented in the

form of an infinite series. Understanding FMA is a topic by itself and more information about the FMA can be found in [29].

Profiling study indicates that the leaf level computations are more intense and load imbalanced than the rest of the computations. Hence, it was decided to load balance this level though load balancing can be done at any stage of the tree. Load imbalance can be attributed to factors like non-uniform distribution of particles, boundary conditions, variation of processor workloads as the simulation proceeds, etc.

For experimental purpose, both uniform and non-uniform particle distribution were used. A non-uniform distribution is any irregular distribution. Two types of non-uniform distribution namely "corner" (corner) and "Gaussian" (Gaussian) were considered. The corner distribution is a special case of Gaussian distribution. It can be obtained by shifting the mean (where the peak of the density occurs) towards one octants of the computational space. The Figure 4.3 and Figure 4.4 represents the corner and the Gaussian distribution respectively. Experiments were conducted on three different sizes of data (20K, 50K, 100k particles). The oct-tree has an height of 4 with 512 leaves. Four multipole terms were used for force calculation.

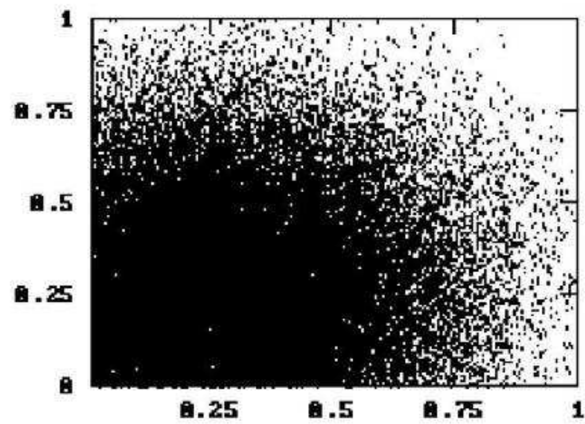


Figure 4.3: Corner distribution

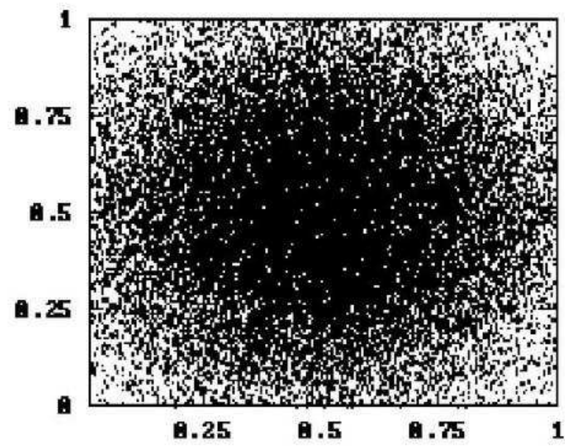


Figure 4.4: Gaussian distribution

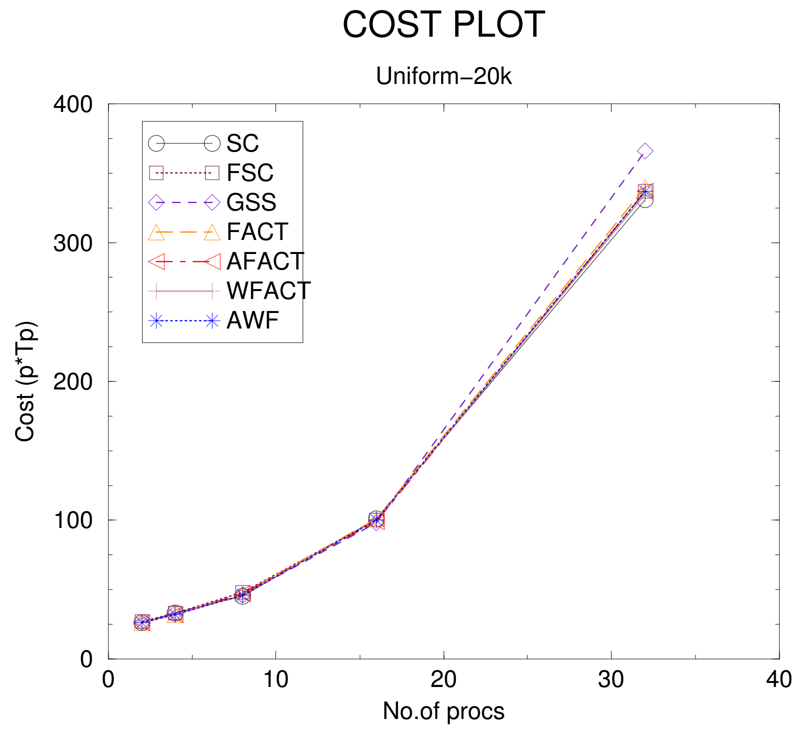


Figure 4.5: Cost: Uniform-20k particles

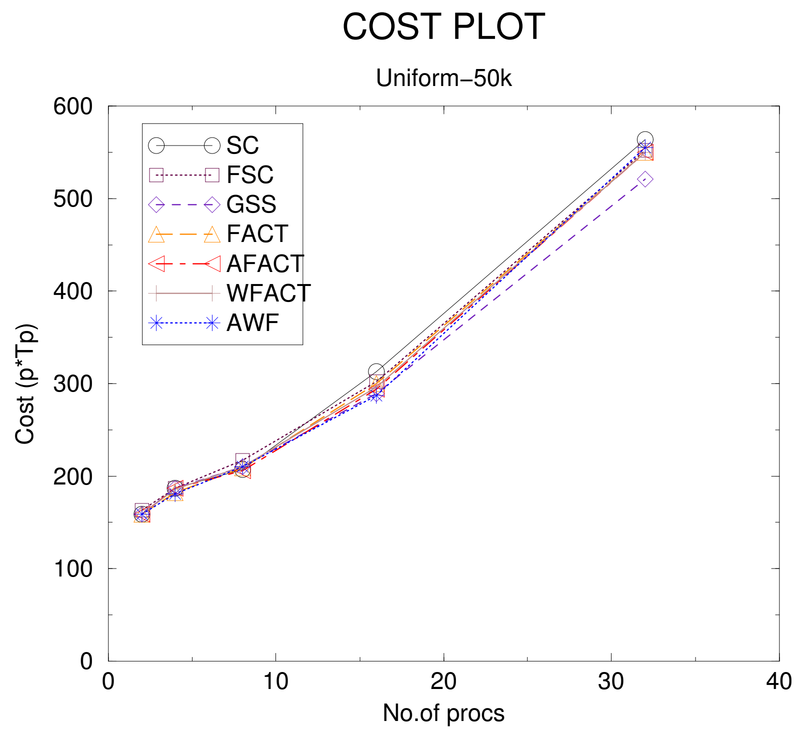


Figure 4.6: Cost: Uniform-50k particles

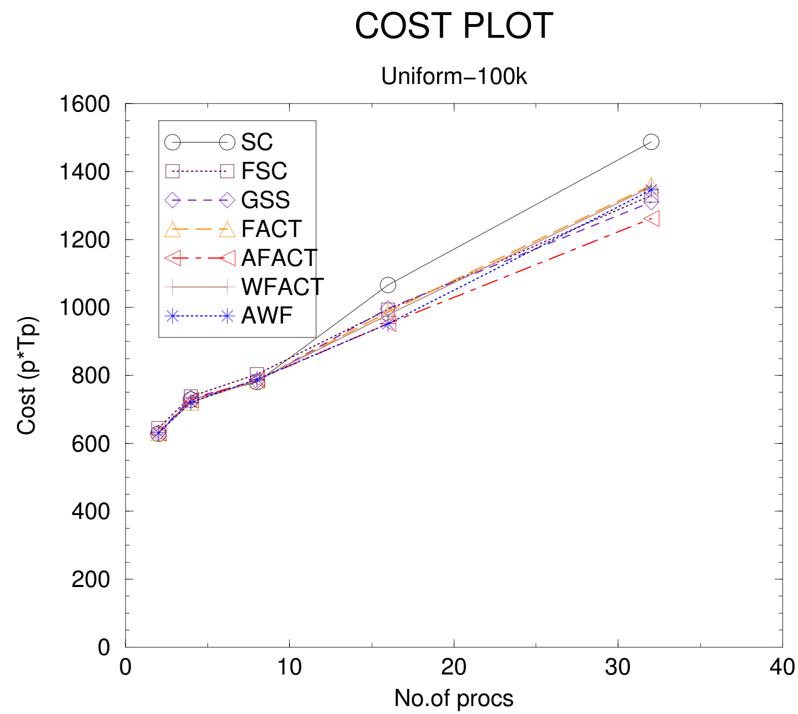


Figure 4.7: Cost: Uniform-100k particles

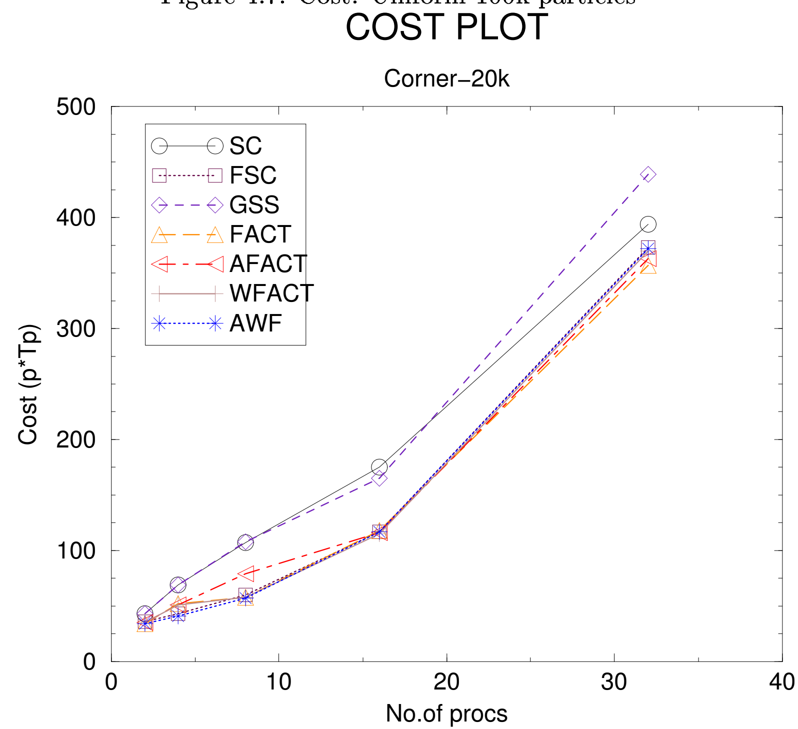


Figure 4.8: Cost: Corner-20k particles

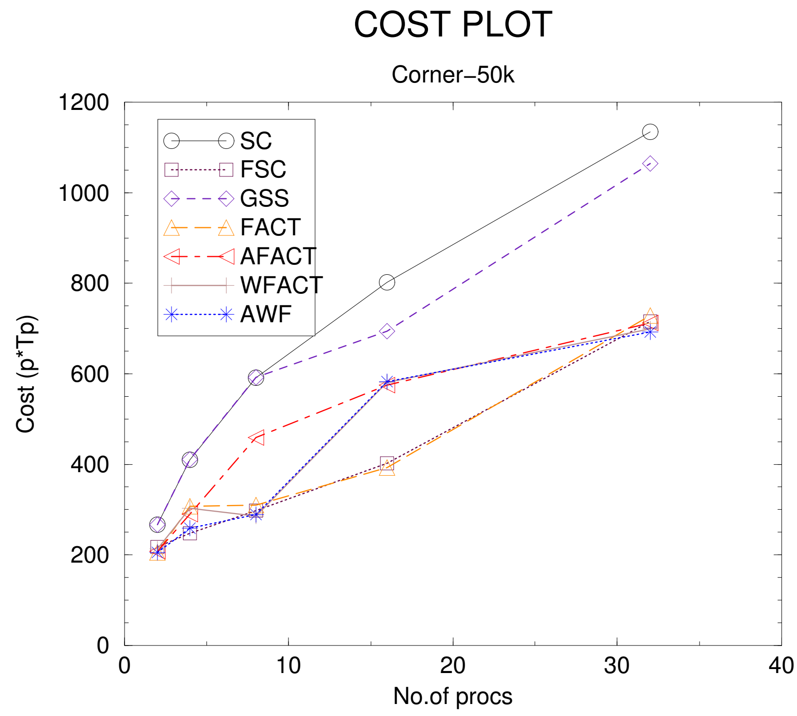


Figure 4.9: Cost: Corner-50k particles

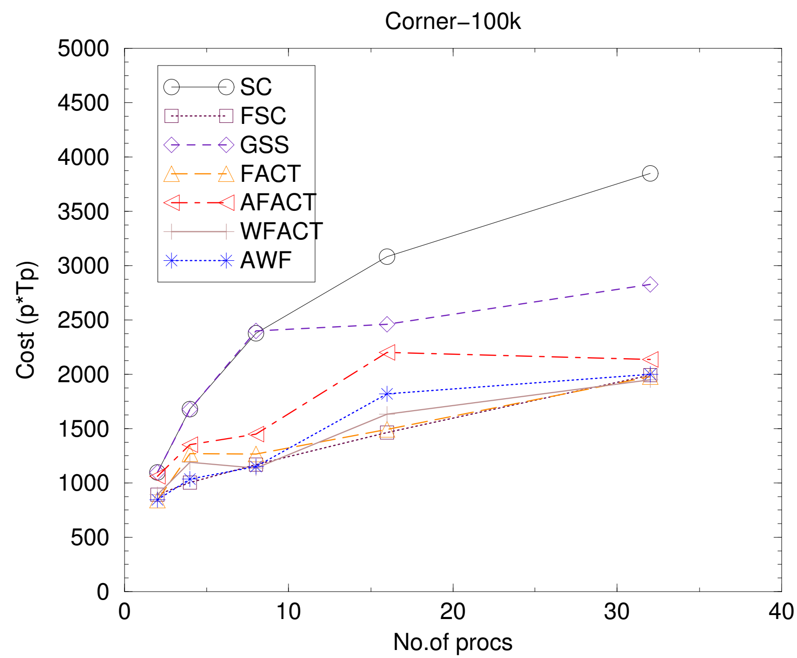


Figure 4.10: Cost: Corner-100k particles

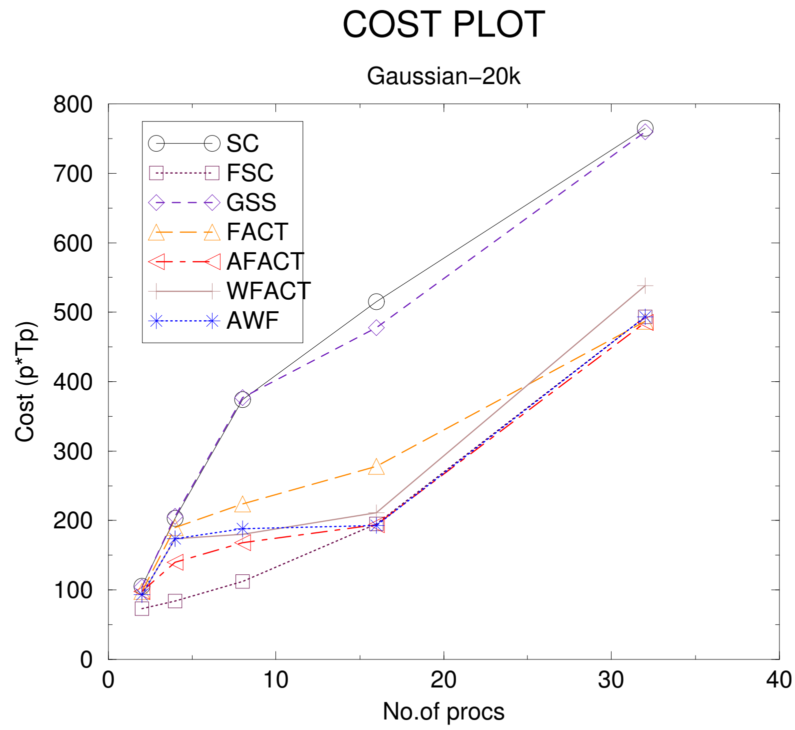


Figure 4.11: Cost: Gaussian-20k particles

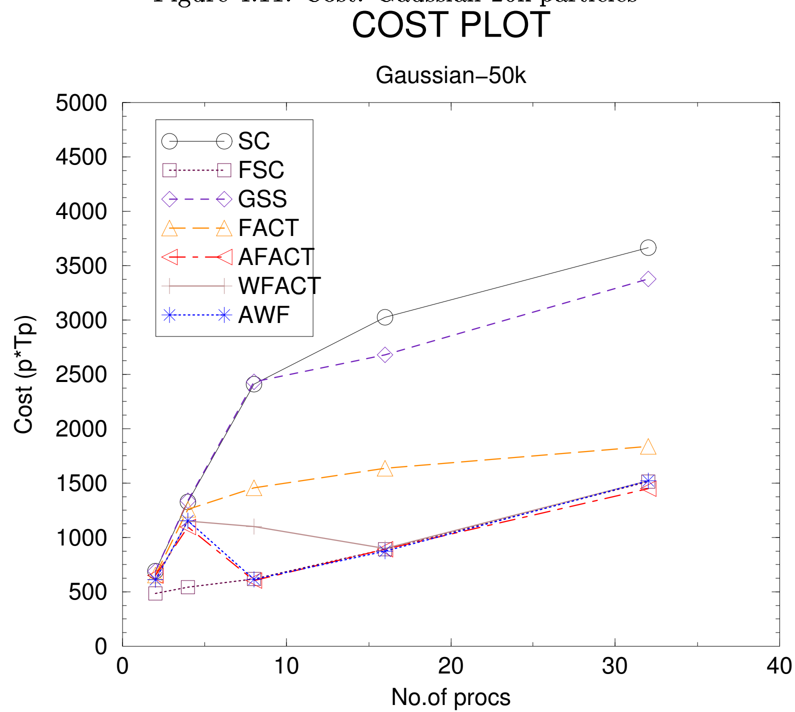


Figure 4.12: Cost: Gaussian-50k particles

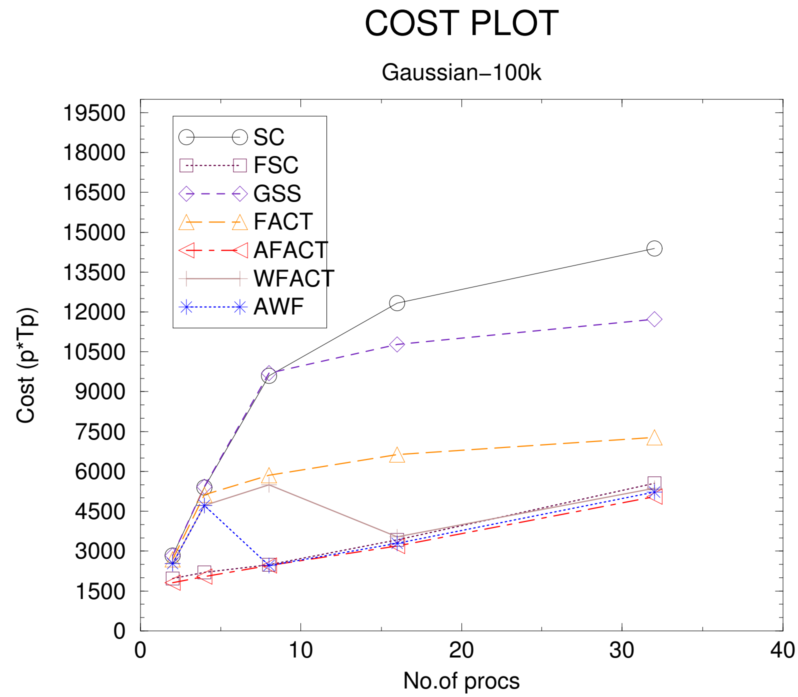


Figure 4.13: Cost: Gaussian-100k particles

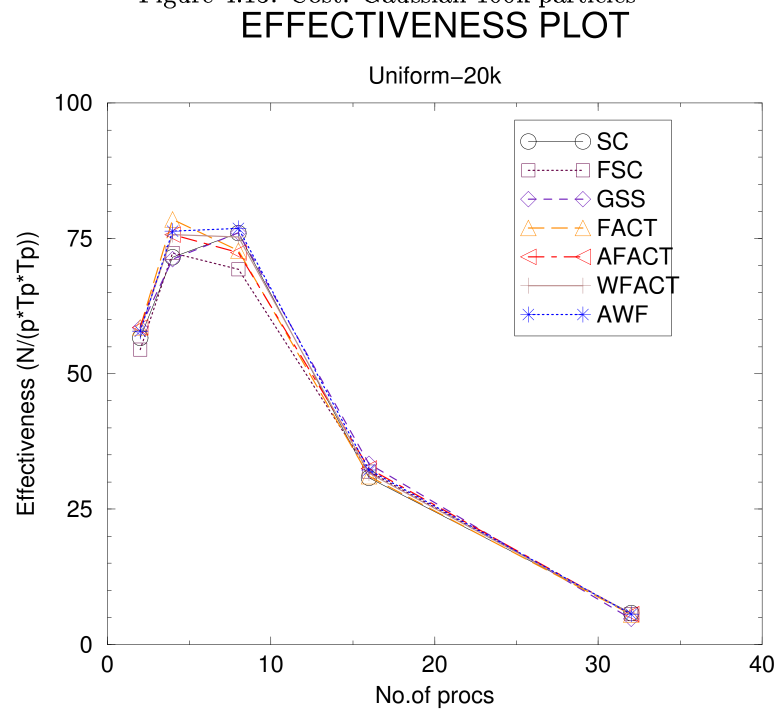


Figure 4.14: Effectiveness: Uniform-20k particles

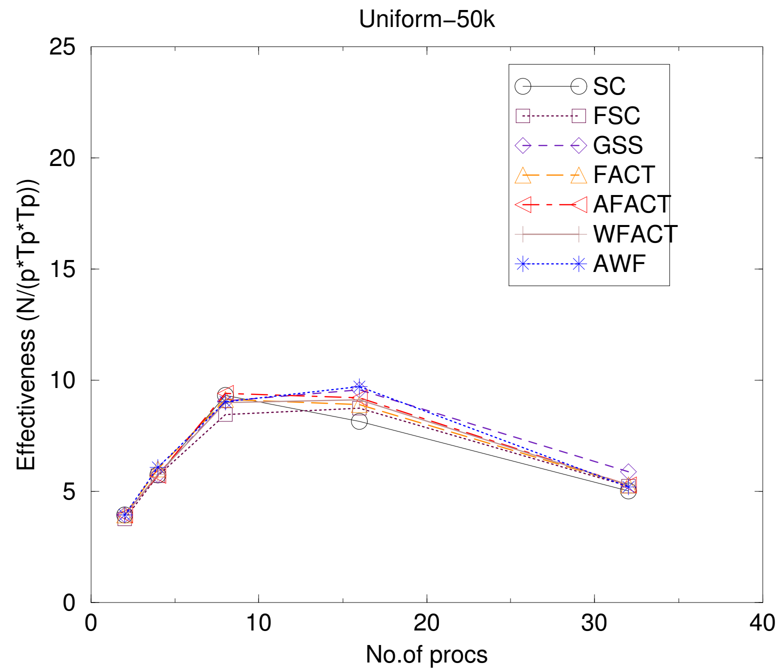


Figure 4.15: Effectiveness: Uniform-50k particles

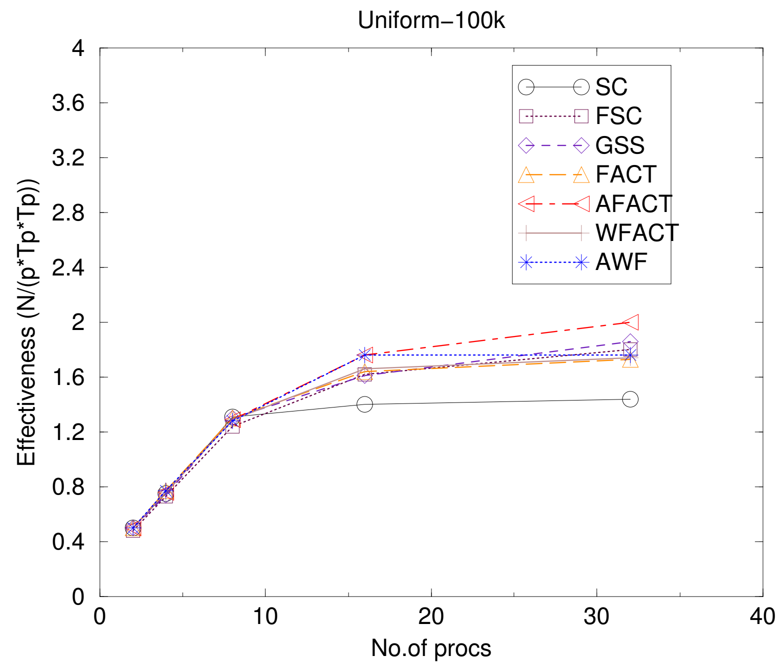


Figure 4.16: Effectiveness: Uniform-100k particles

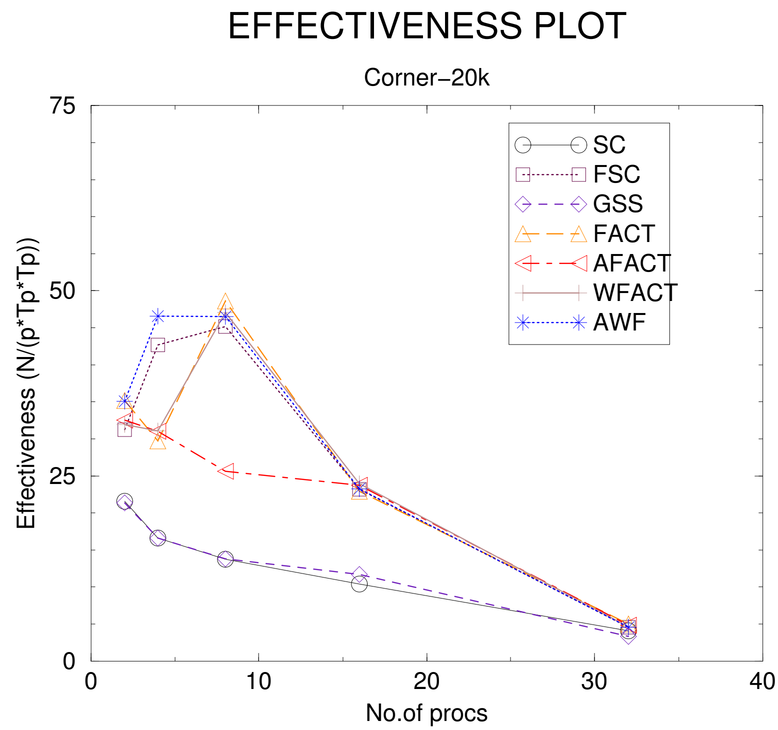


Figure 4.17: Effectiveness: Corner-20k particles

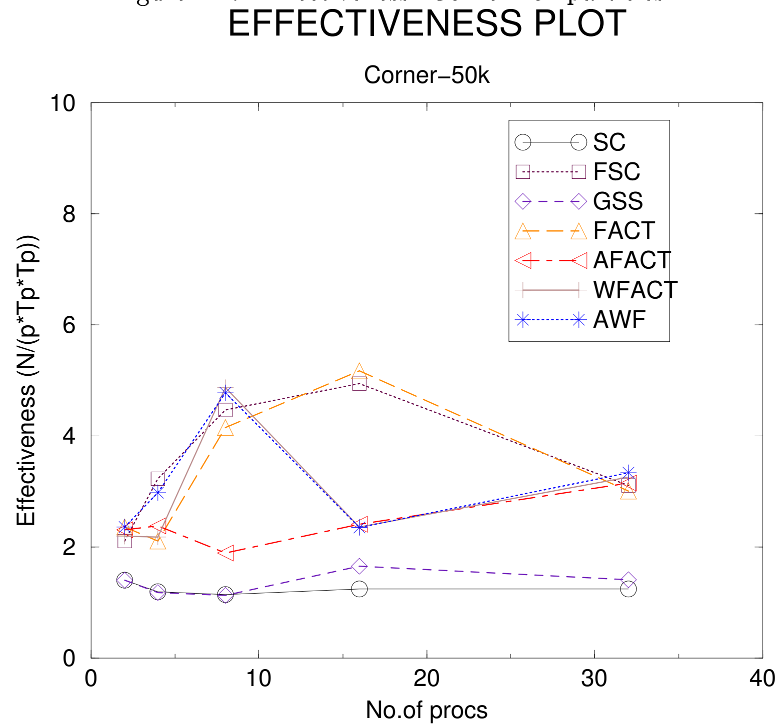


Figure 4.18: Effectiveness: Corner-50k particles

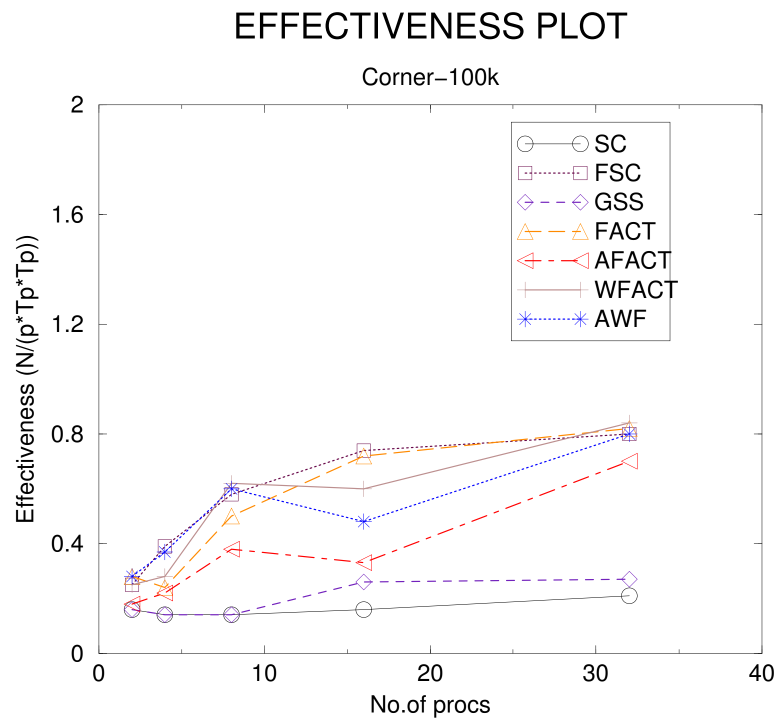


Figure 4.19: Effectiveness: Corner-100k particles

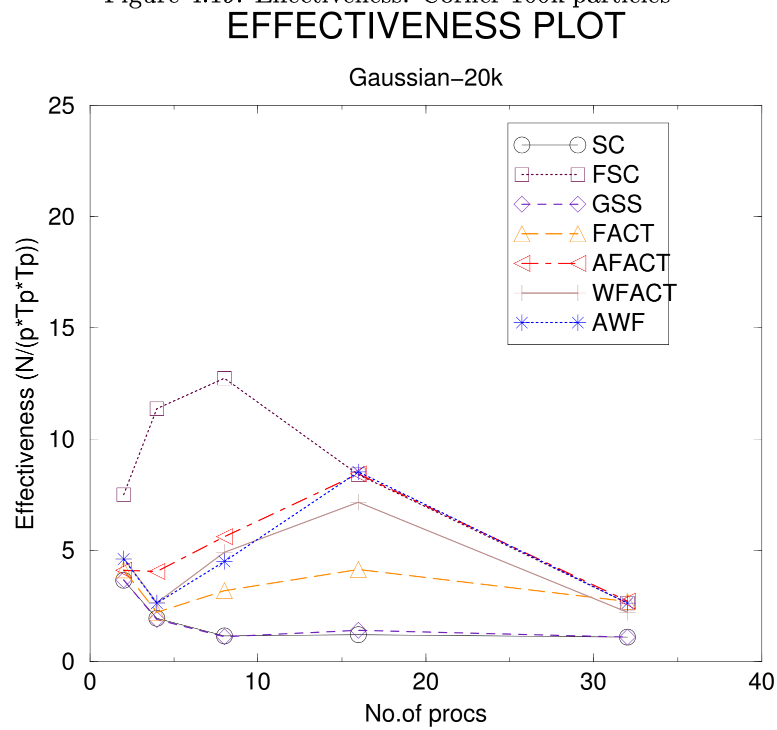


Figure 4.20: Effectiveness: Gaussian-20k particles

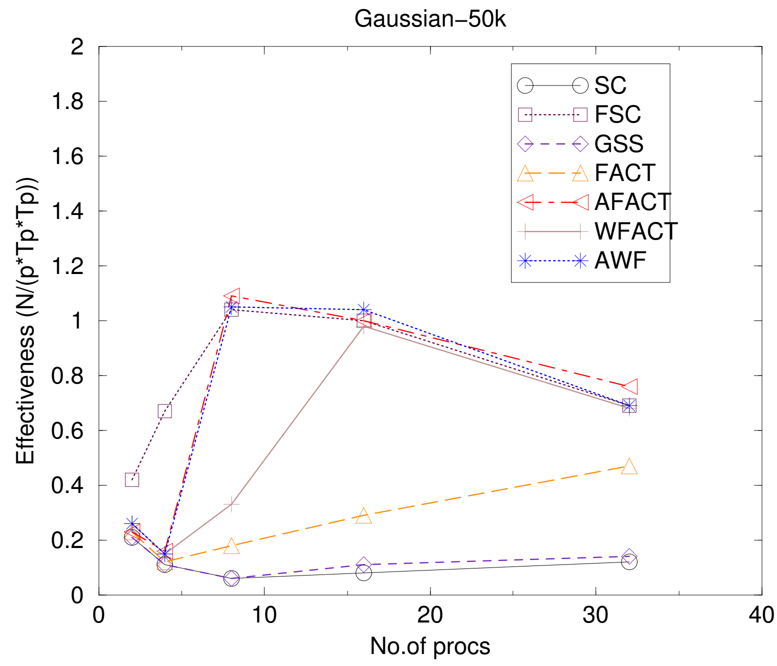


Figure 4.21: Effectiveness: Gaussian-50k particles

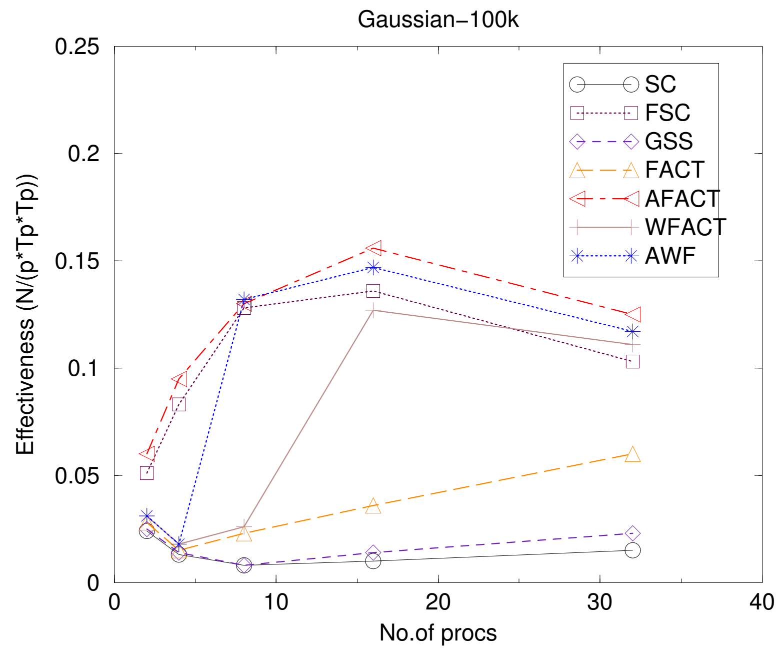


Figure 4.22: Effectiveness: Gaussian-100k particles

Table 4.2: Best case C.O.V for $N$-body simulations I

| | SC | FSC | GSS | FACT |
|---|---|---|---|---|
| u20k | 0.0062-2p | 0.0167-4p | 0.0034-2p | 0.0013-2p |
| c20k | 0.1579-32p | 0.0323-8p | 0.1346-32p | 0.0133-2p |
| g20k | 0.4976-32p | 0.0946-8p | 0.3889-32p | 0.2204-32p |
| u50k | 0.0002-2p | 0.0203-4p | 0.0009-2p | 0.0015-2p |
| c50k | 0.4227-2p | 0.0438-4p | 0.2731-32p | 0.006-2p |
| g50k | 0.9664-2p | 0.1179-8p | 0.9642-2p | 0.5361-32p |
| u100k | 0.0003-2p | 0.0162-8p | 0.00008-2p | 0.0010-2p |
| c100k | 0.4359-2p | 0.0471-4p | 0.3427-32p | 0.0082-2p |
| g100k | 0.9644-2p | 0.1198-4p | 0.9646-2p | 0.7160-32p |

Table 4.3: Best case C.O.V for $N$-body simulations II

| | AFACT | WFACT | AWF |
|---|---|---|---|
| u20k | 0.00093-2p | 0.0027-2p | 0.0029-2p |
| c20k | 0.0684-16p | 0.0378-4p | 0.0021-2p |
| g20k | 0.1480-16p | 0.1696-16p | 0.1634-16p |
| u50k | 0.0015-2p | 0.0044-2p | 0.0017-2p |
| c50k | 0.0241-2p | 0.0549-8p | 0.0067-2p |
| g50k | 0.1225-8p | 0.1964-16p | 0.0839-8p |
| u100k | 0.0024-2p | 0.0035-2p | 0.0041-2p |
| c100k | 0.1289-32p | 0.0564-8p | 0.0101-2p |
| g100k | 0.01518-4p | 0.2166-16p | 0.1106-8p |

Table 4.4: % cost improvement over static chunking for $N$-body simulations

| | FSC | GSS | FACT | AFACT | WFACT | AWF |
|---|---|---|---|---|---|---|
| u20k | 0.99-16p | 2.97-16p | 3.03-4p | 3.03-4p | 3.03-4p | 3.03-4p |
| u50k | 3.51-16p | 7.66-16p | 4.15-16p | 6.07-16p | 5.43-16p | 8.30-16p |
| u100k | 10.55-32p | 11.90-32p | 8.54-32p | 15.13-32p | 8.87-32p | 10.69-16p |
| c20k | 43.92-8p | 5.71-16p | 45.79-8p | 33.71-16p | 45.79-8p | 46.72-8p |
| c50k | 49.87-16p | 13.46-16p | 50.99-16p | 37.26-32p | 51.60-8p | 51.09-8p |
| c100k | 52.59-16p | 26.55-32p | 51.65-16p | 44.49-32p | 52.18-8p | 51.59-8p |
| g20k | 70.05-8p | 7.18-16p | 46.01-16p | 62.33-16p | 59.02-16p | 62.52-16p |
| g50k | 74.32-8p | 11.43-16p | 49.82-32p | 74.98-8p | 70.26-16p | 74.53-8p |
| g100k | 74.02-8p | 18.46-32p | 49.41-32p | 74.02-8p | 71.30-16p | 74.37-8p |

The application suffers little load imbalance due to uniform distribution of the particles in the computational space. However, the application may suffer from load imbalances due to algorithmic variance, systemic variance, etc. The *cost* plot shows that there is no substantial performance difference between the scheduling techniques. Nonetheless, all the techniques perform better than static chunking (without load balancing). When the problem size is scaled up, the performance difference between static chunking and other techniques widens. Adaptive factoring gave the maximum cost improvement over static chunking (up to 15.13 %) for 100k problem size.

The application experiences significant load imbalance because of non-uniform (corner and Gaussian) distribution of the particles in the computational space. All the scheduling techniques consistently outperformed static chunking with the scaling up of the problem size. Techniques like guided self scheduling, factoring tend to perform poor when more work is allocated during the earlier chunks for the reason during the later stages, there wont be enough work to smooth out the load imbalance caused by the earlier chunks. In both Gaussian and corner distribution, major work is present in the front, though in corner distribution, the effect is little less. Thats

the reason why guided self scheduling performed worser than all the other techniques for these distributions and factoring, the second worst technique for Gaussian distribution. For corner and Gaussian distribution, factoring gave up to 50% cost improvement over static chunking. Since, the optimal chunk size was empirically determined, fixed size chunking gave the best performance for non-uniform distributions. For corner distribution, fixed size chunking gave up to 52% cost improvement over static chunking and for Gaussian distribution gave up to 74% cost improvement over static chunking.

Among the variants of factoring based methods, for corner distribution, weighted factoring and adaptive weighted factoring seem to perform as good as factoring (gave up to 51% cost improvement over static chunking). However, for Gaussian distribution, they outperformed factoring (gave up to 70% cost improvement over static chunking compared to 50% by factoring) with adaptive weighted factoring performing marginally better than weighted factoring (up to 74% cost improvement for adaptive weighted factoring compared to 71% cost improvement for weighted factoring). Adaptive factoring performed very well for Gaussian distribution along with fixed size chunking (up to 74% cost improvement over static chunking). For corner distribution, adaptive factoring does not seem to perform well (gave up to 45% cost improvement over static chunking) as well as other factoring based methods. Table 4.2 and 4.3 shows the coefficient of variation (c.o.v) of processor finishing times of various scheduling techniques. Shown in the tables are the best case c.o.v of each technique. For non-uniform distributions like corner and Gaussian distribution, the c.o.v of all techniques are consistently smaller than static chunking. This shows that how effectively the techniques have utilized the processing resource.

### 4.6 Automatic Quadrature Routine

This application was developed by [30]. The application involves generating the profile of an automatic quadrature routine (AQR) which is a computationally intensive and embarrassingly parallel task. The quadrature routine ADLEV [31] was selected for profiling purpose. The quadrature routine approximates an integral of the form $I = \int f(x)\ dx$, where D is the domain of the integration, and f(x) is the integrand. In 1-dimension, the integral f(x) dx is approximated as $\sum_{(i=1)}^{N} w_i f(x_i)$ where $x_i$ and $w_i$ are the abscissae and the weights of the quadrature rule respectively. The routine takes (i) a description of the domain D, (ii) the code for the integrand

f(x), (iii) absolute and relative error tolerances, (iv) a limit to the number of function evaluations in case the error tolerances are not achievable by the AQR, and (v) the quadrature rule to be used as input parameters. The outputs from the routine are: (i) approximation to the integral, (ii) an estimate of the absolute error, (iii) the actual number of function evaluations used, and (iv) a termination condition indicator. Typically, the integrands are chosen such that the answers are known analytically to facilitate the computation of the true error of the computed integral, and the accuracy of the error estimate.

Suppose, if DATA is an input array which stores information about the set of parameters for the quadrature routine which includes: (i) the type of integrand function, (ii) difficulty level, (iii) the accuracy requirements, (iv) the quadrature rule settings, and (v) the number of sample integrals to be evaluated for the parameter set, then the sequential algorithm can be expressed as:

```
for i=0 to N-1 do
        Evaluate integrals specified by DATA(i)
        Compute accuracy and cost
        Compute statistics related to quadrature routine
end for
```

Since, all the calculations associated with DATA(i) is a single computational task, the profiling algorithm can be considered as a parallel loop with N independent iterates.

The integrand functions can be broadly classified as follows: well-behaved, oscillatory, C0 function, Gaussian peak, internal peak, singular at an interior point, or singular along the edge. The singular integrand and the internal peak integrand are the two-most time consuming integrand family. By placing the time consuming iterates in four different places, four different distributions can be generated. They are front-heavy, back-heavy, center-heavy, and scatter-heavy distribution. In front-heavy distribution, the singular integrands are evaluated first followed by the evaluation of the easy integrands. For back-heavy distribution, the order is reversed. In center-heavy distribution, the most time consuming integrands appear in the middle of the loop. Finally, for scatter-heavy distribution, the time consuming integrands are scattered throughout the loop. For experimental purpose, three different N's (3780,7560,15120) were chosen. For each N, four different distributions were generated.

The following figures gives an idea of how the iteration execution times are distributed for different distributions.

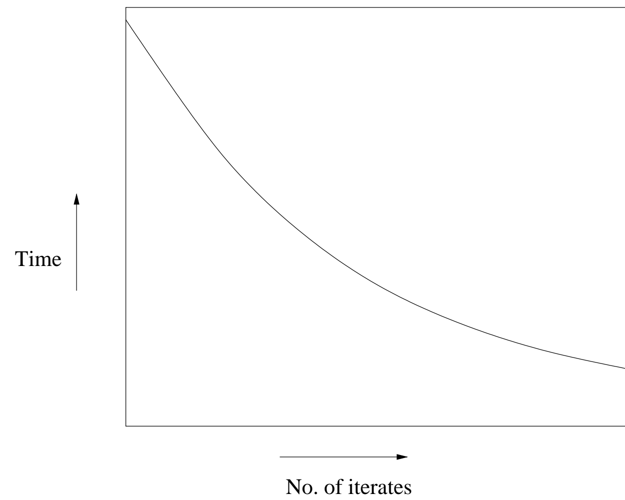


Figure 4.23: Front distribution

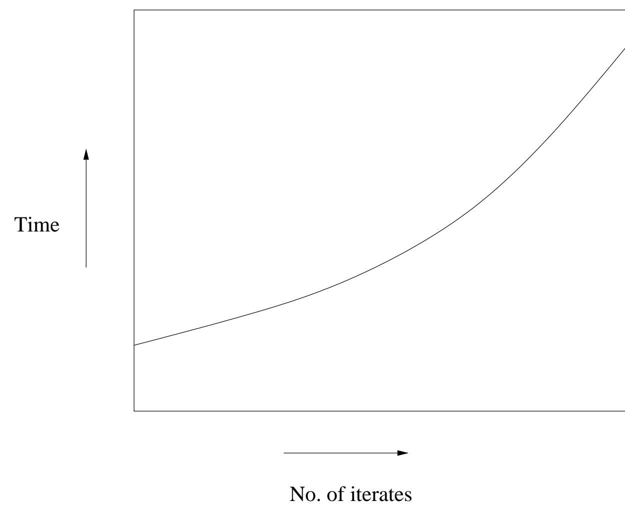


Figure 4.24: Back distribution

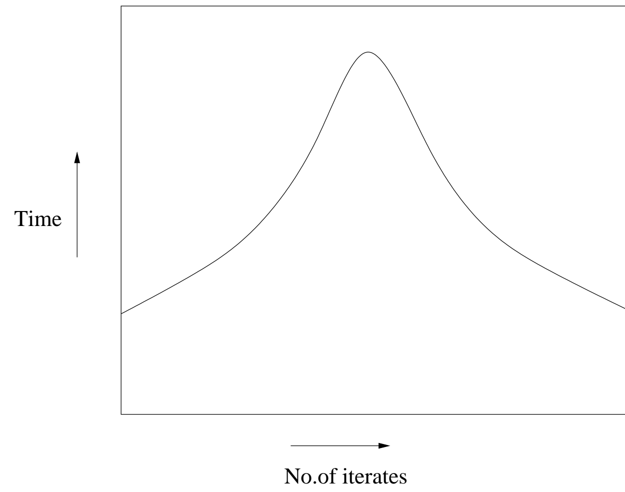


Figure 4.25: Center distribution

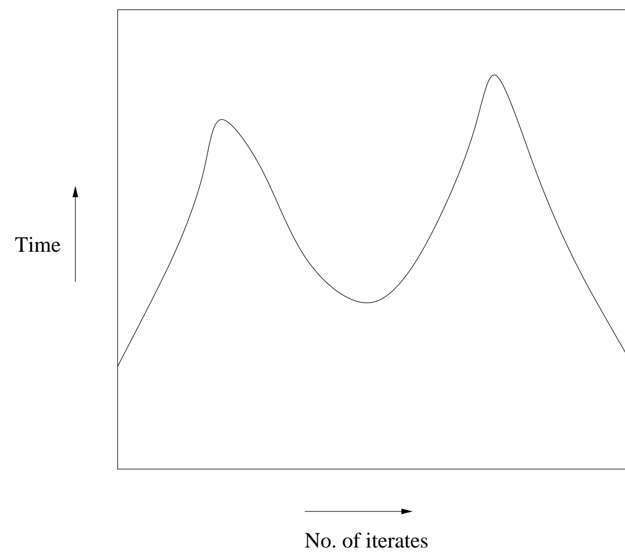


Figure 4.26: Scatter distribution

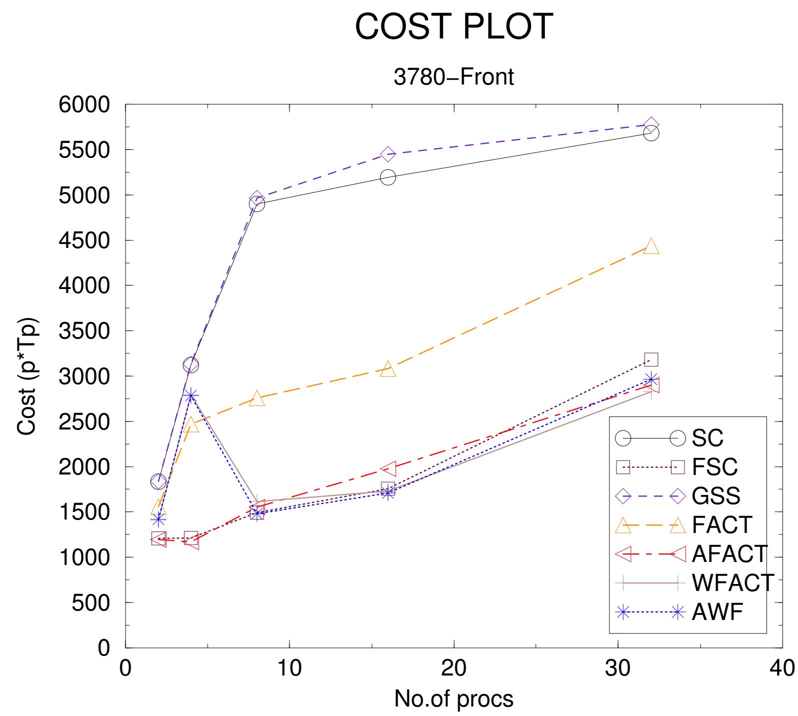


Figure 4.27: Cost: 3780-Front

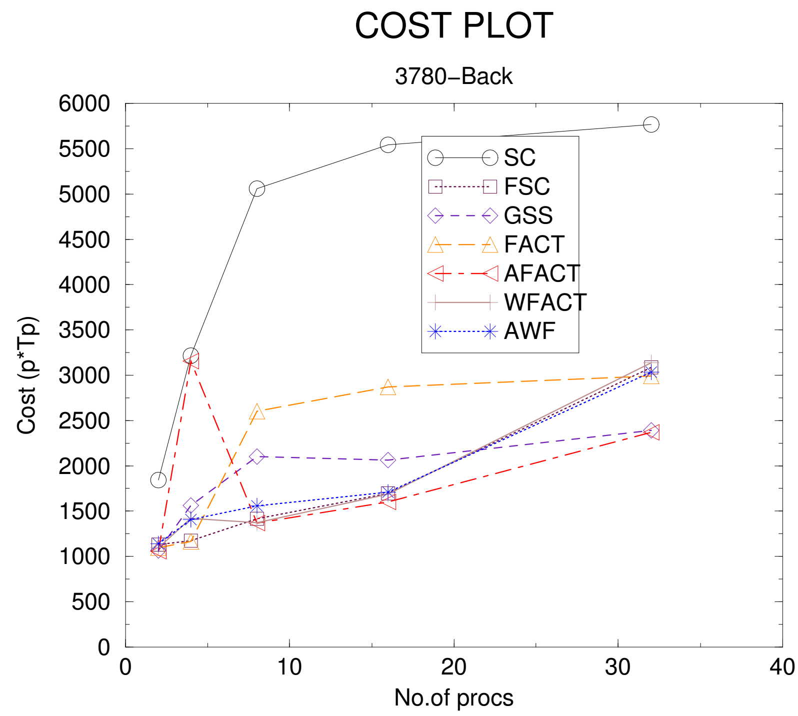


Figure 4.28: Cost: 3780-Back

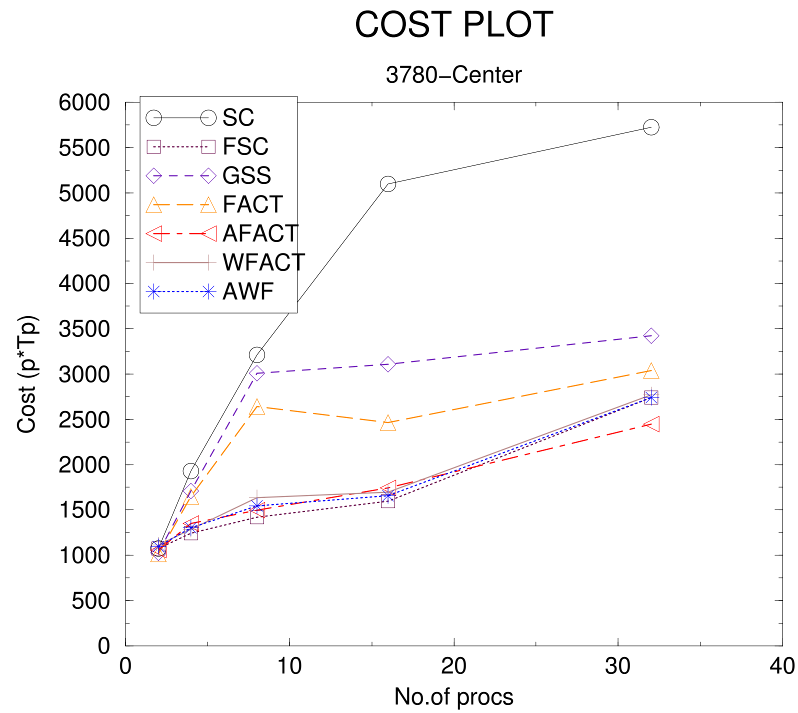


Figure 4.29: Cost: 3780-Center

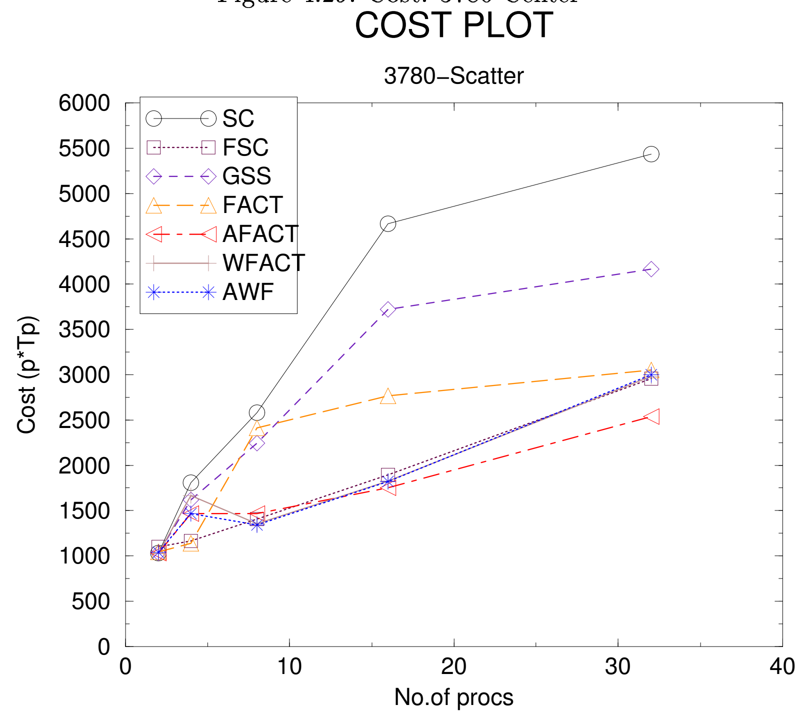


Figure 4.30: Cost: 3780-Scatter

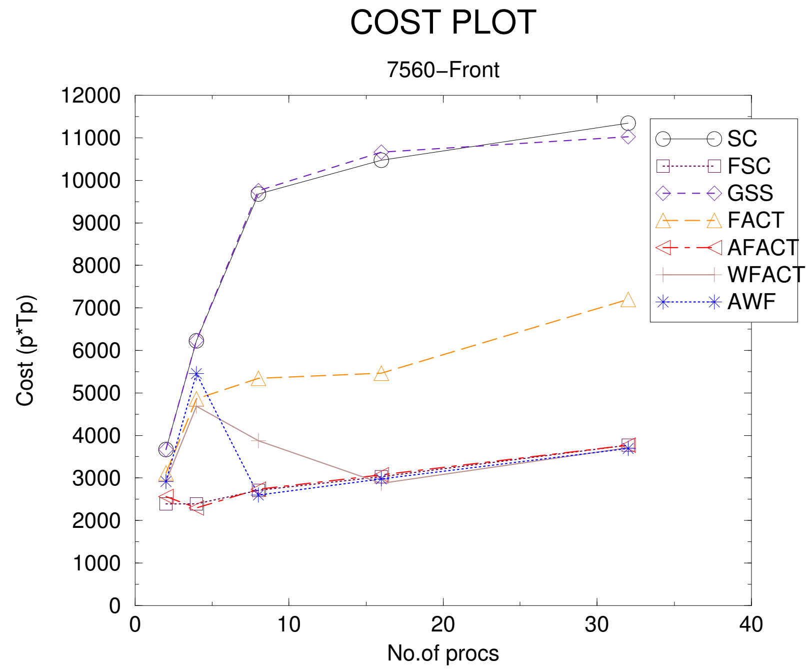


Figure 4.31: Cost: 7560-Front

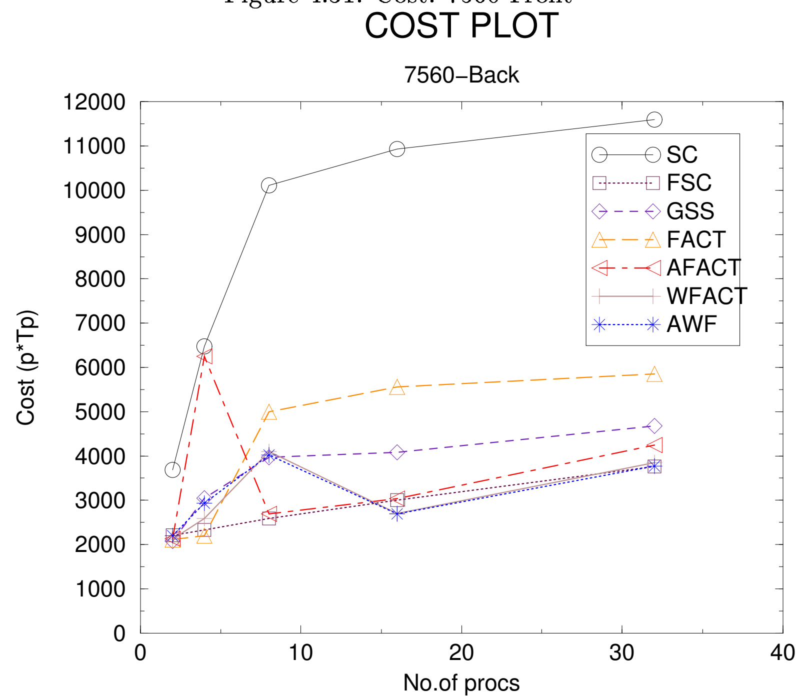


Figure 4.32: Cost: 7560-Back

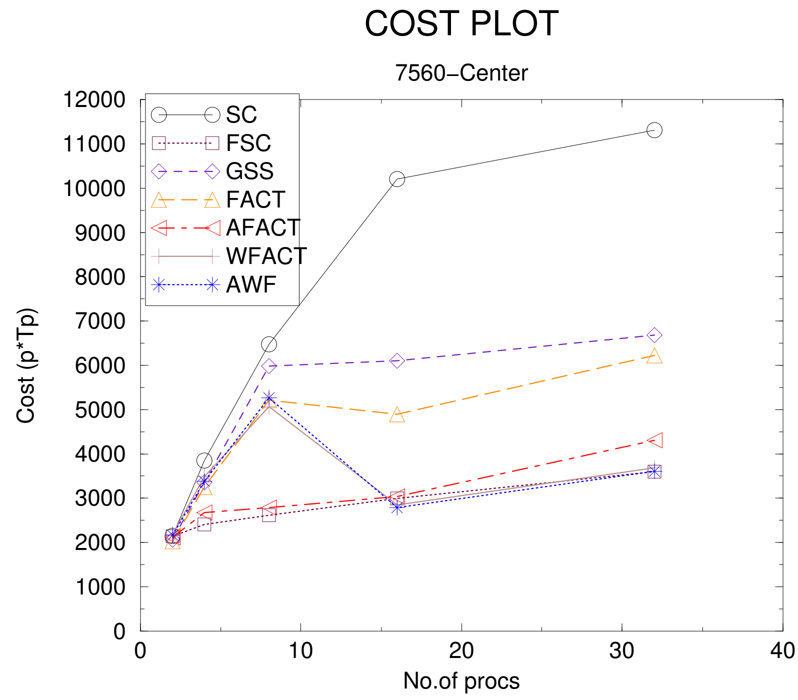


Figure 4.33: Cost: 7560-Center

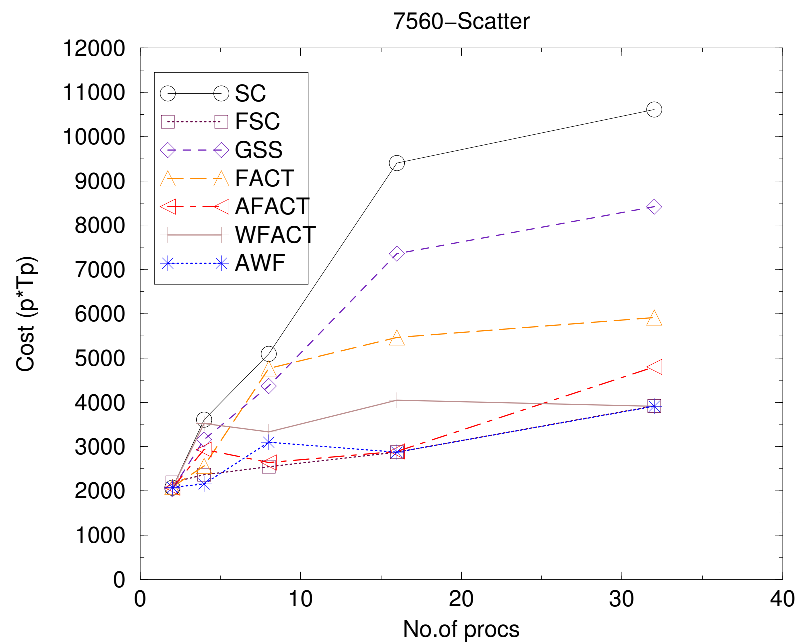


Figure 4.34: Cost: 7560-Scatter

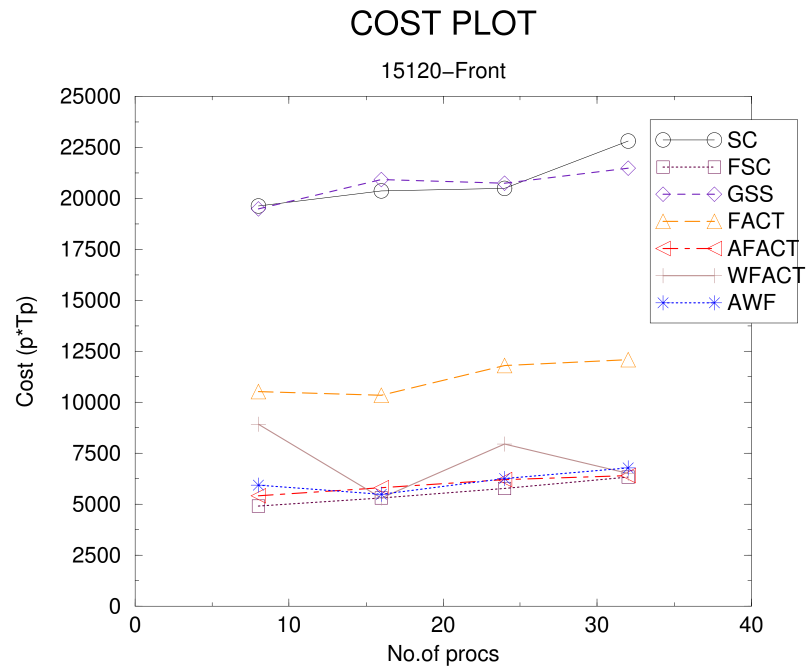


Figure 4.35: Cost: 15120-Front

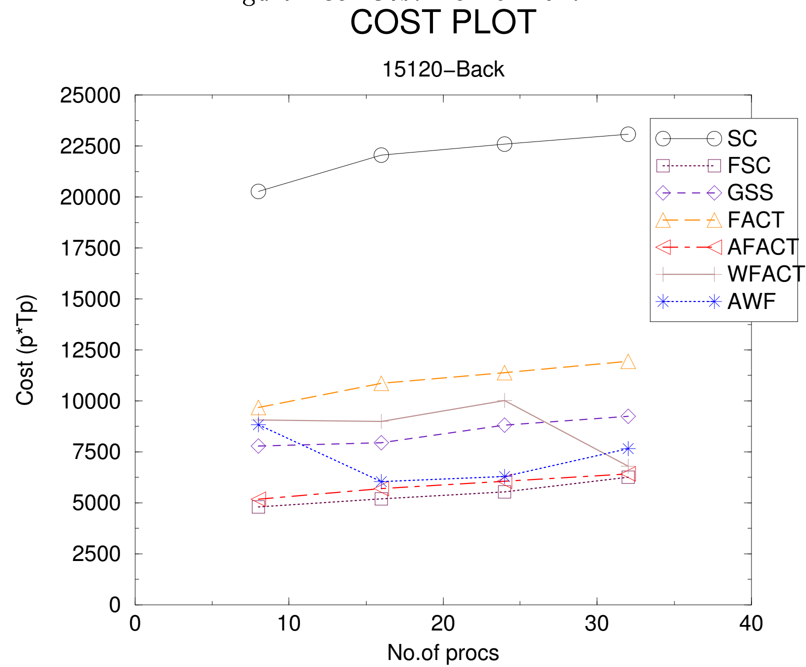


Figure 4.36: Cost: 15120-Back

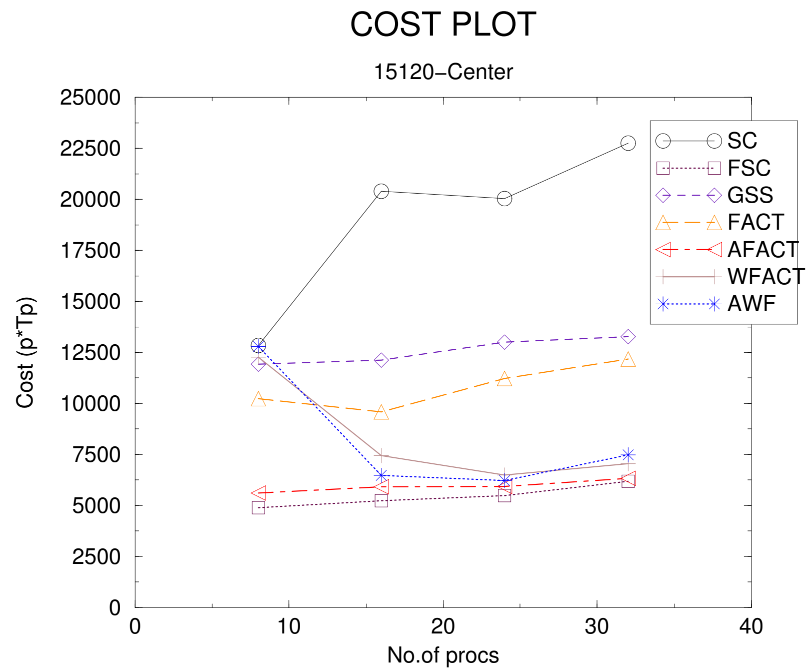


Figure 4.37: Cost: 15120-Center

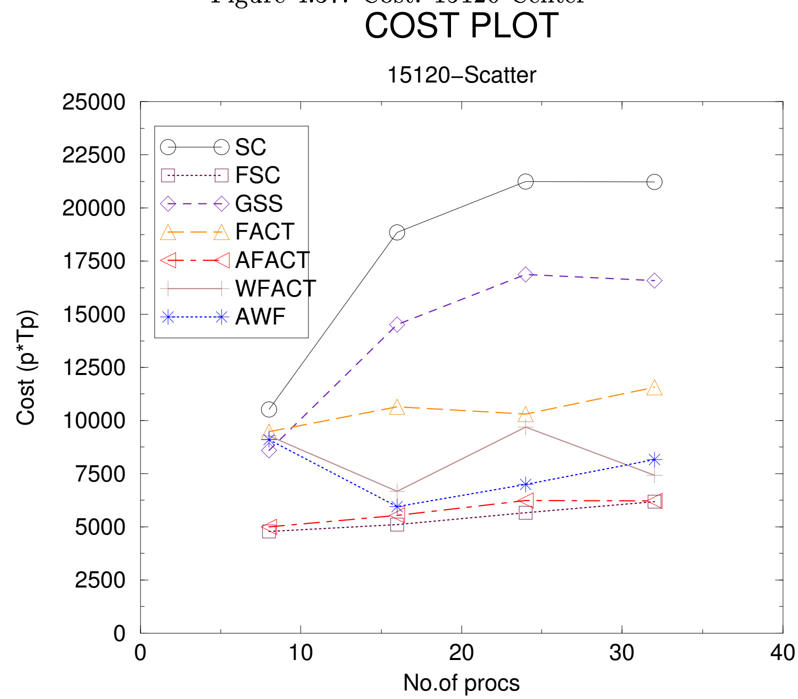


Figure 4.38: Cost: 15120-Scatter

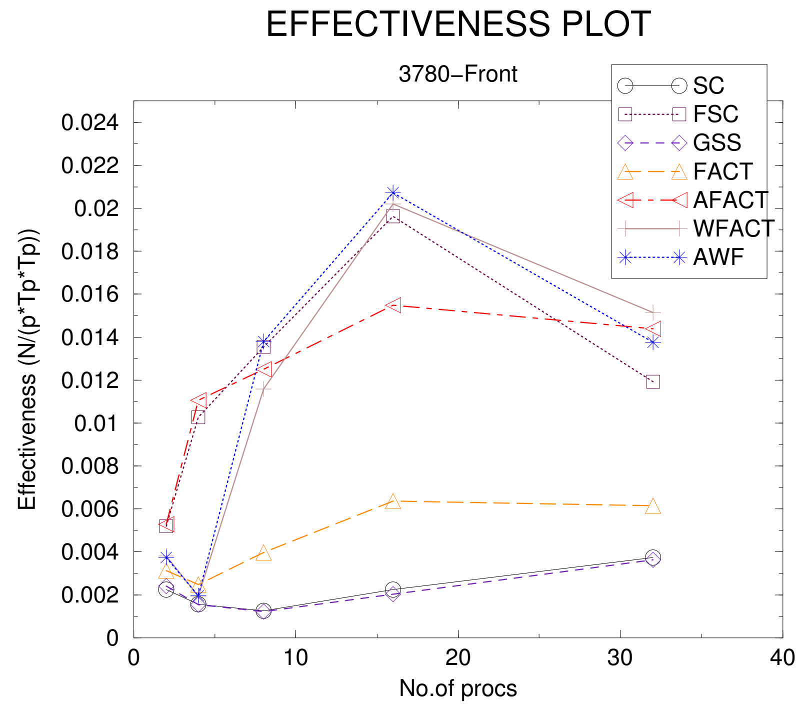


Figure 4.39: Effectiveness: 3780-Front

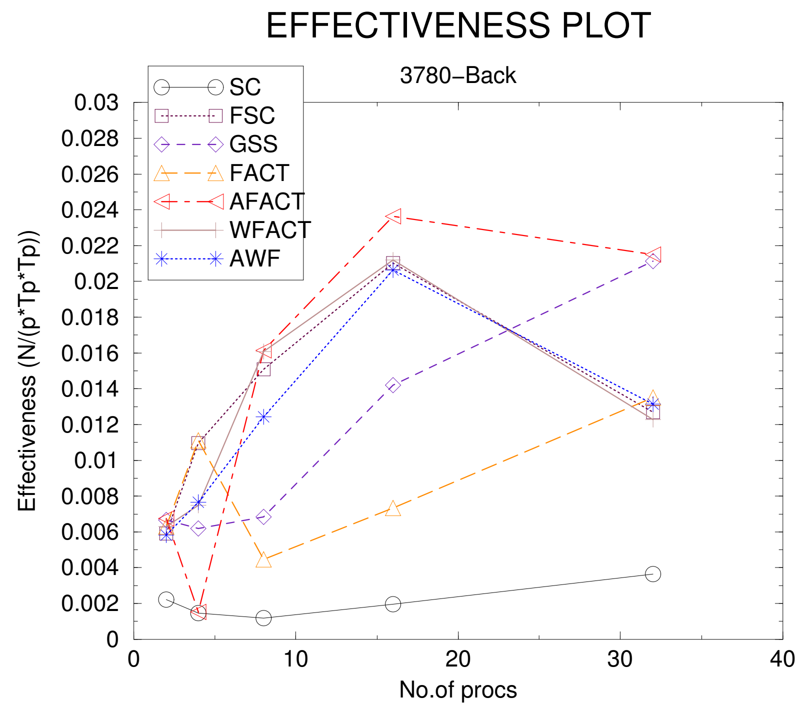


Figure 4.40: Effectiveness: 3780-Back

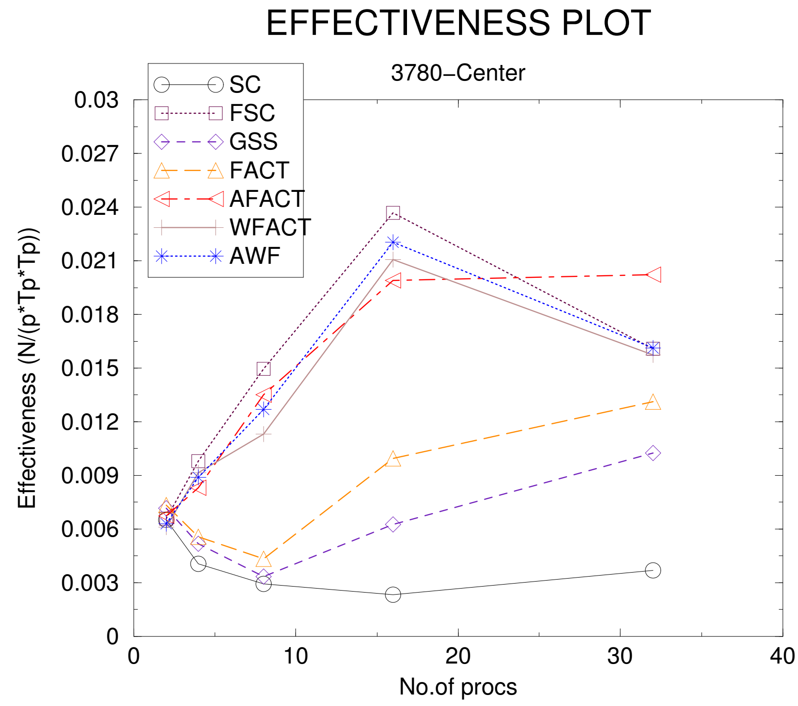


Figure 4.41: Effectiveness: 3780-Center

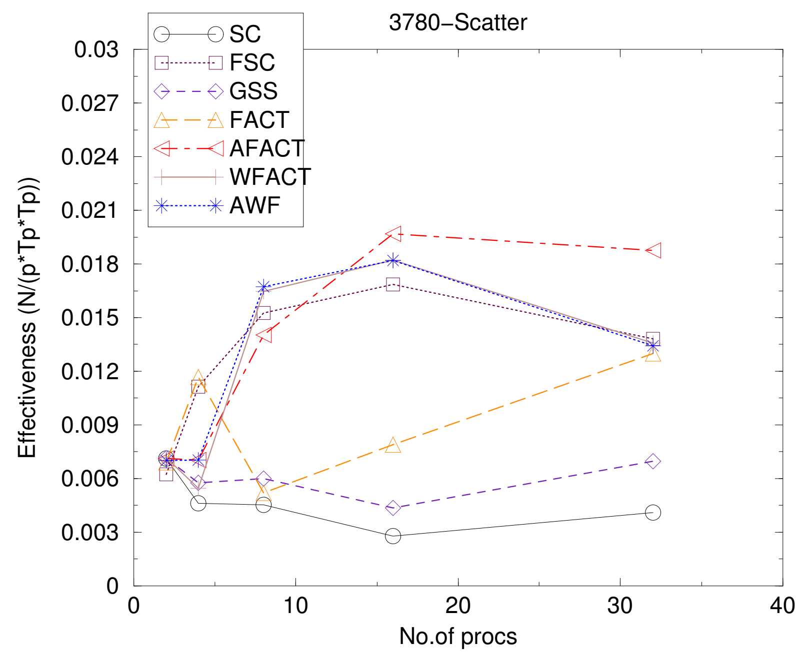


Figure 4.42: Effectiveness: 3780-Scatter

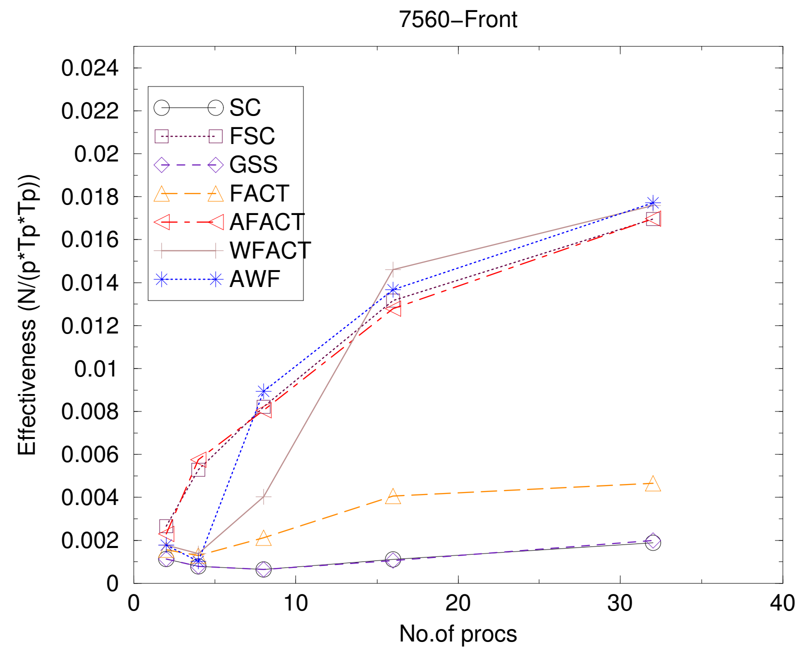


Figure 4.43: Effectiveness: 7560-Front

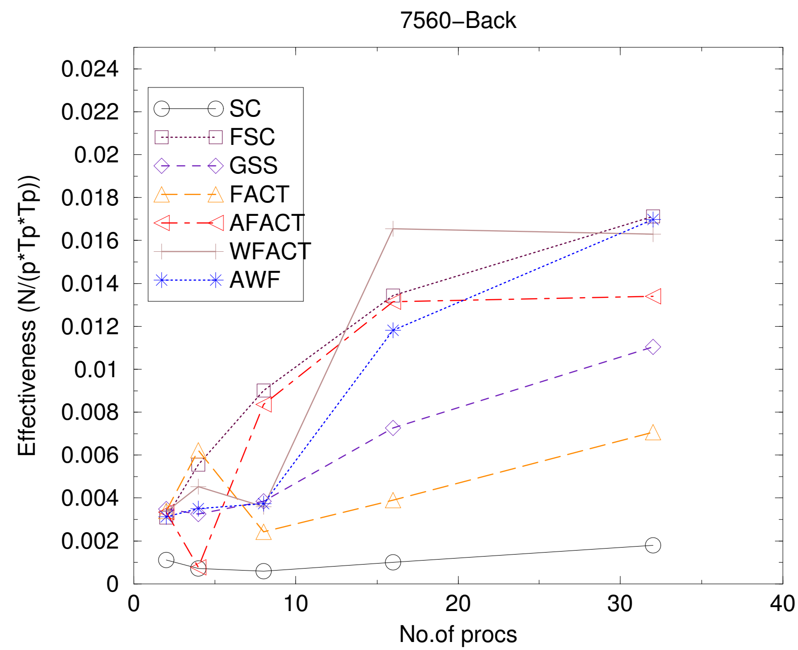


Figure 4.44: Effectiveness: 7560-Back

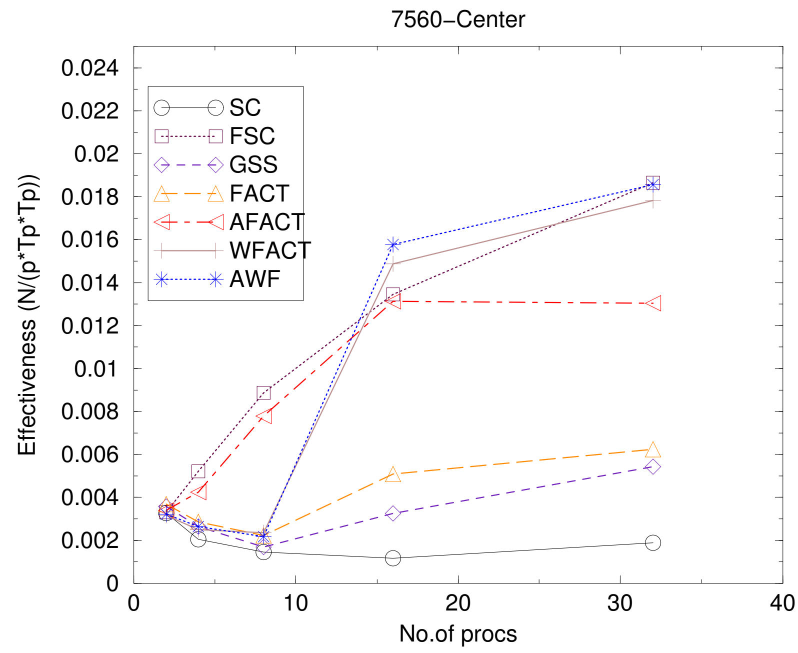


Figure 4.45: Effectiveness: 7560-Center

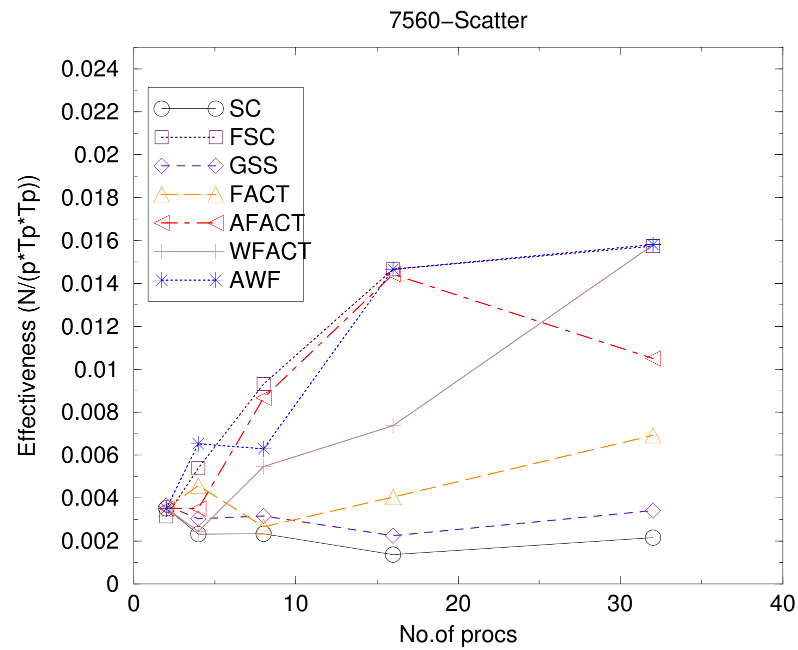


Figure 4.46: Effectiveness: 7560-Scatter

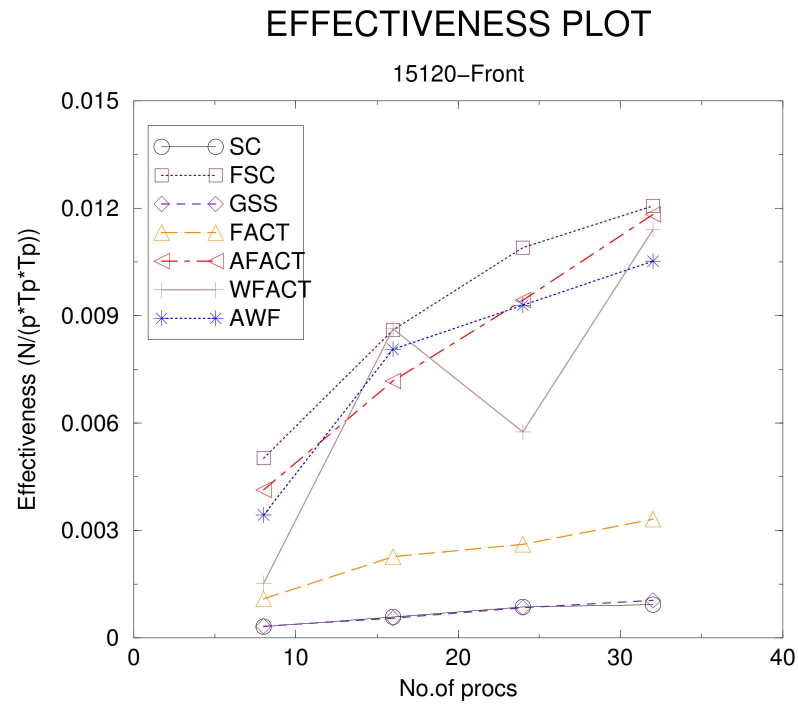


Figure 4.47: Effectiveness: 15120-Front

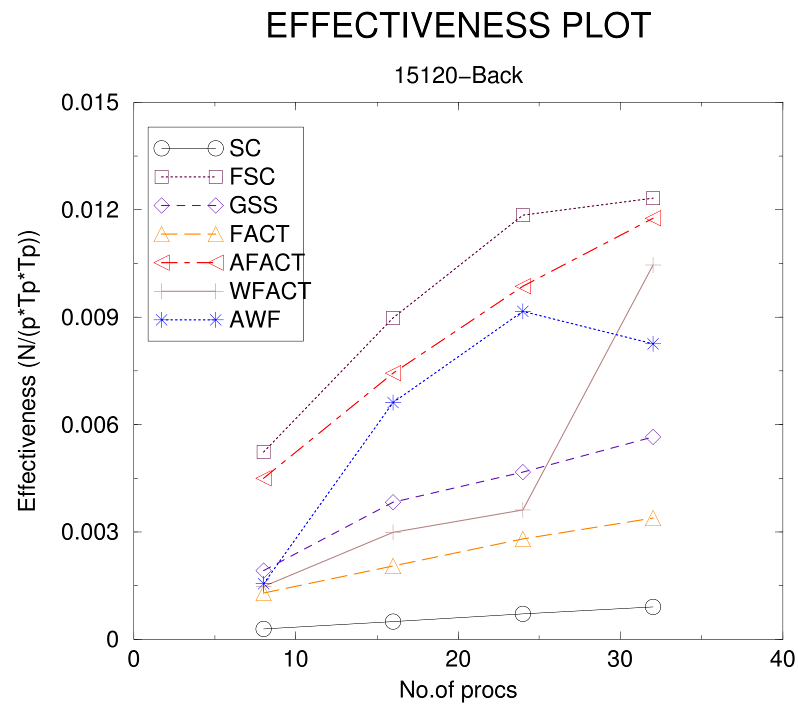


Figure 4.48: Effectiveness: 15120-Back

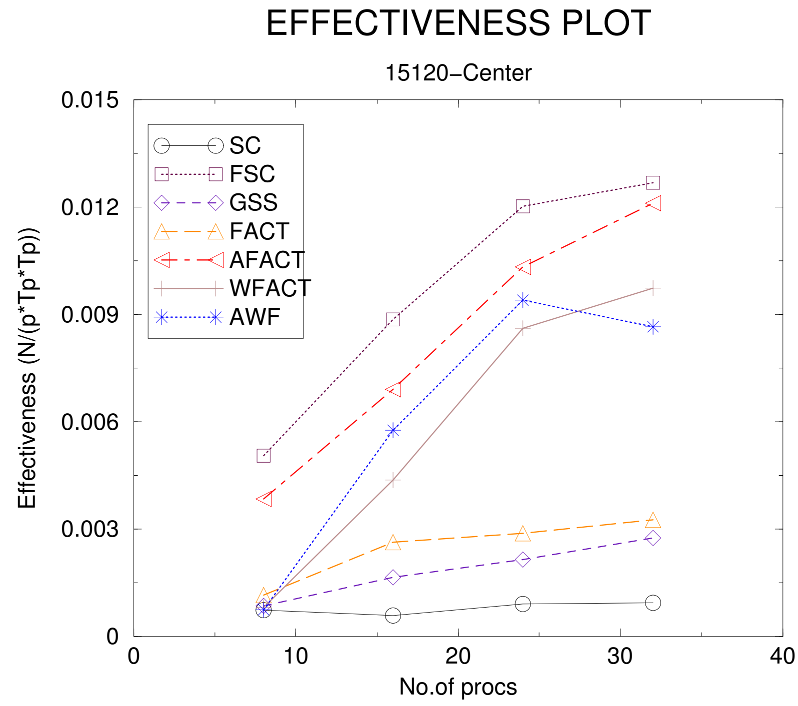


Figure 4.49: Effectiveness: 15120-Center

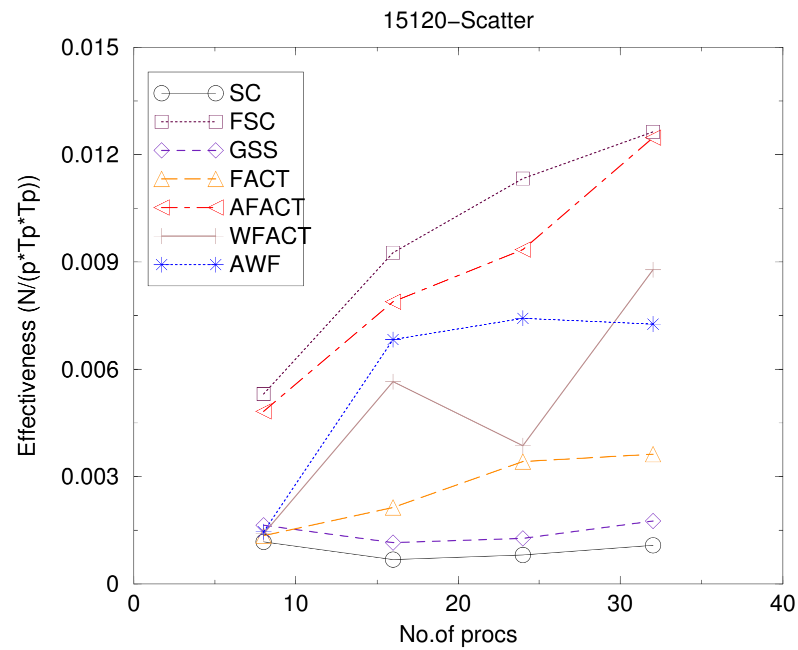


Figure 4.50: Effectiveness: 15120-Scatter

Table 4.5: Best case C.O.V for automatic quadrature routine I

| | SC | FSC | GSS | FACT |
|---|---|---|---|---|
| 3780-F | 1.161530-2p | 0.109988-4p | 1.162153-2p | 0.680928-4p |
| 3780-B | 1.163107-2p | 0.030263-2p | 0.013359-2p | 0.004928-2p |
| 3780-C | 0.089716-2p | 0.057293-2p | 0.021969-2p | 0.003127-2p |
| 3780-S | 0.032133-2p | 0.040636-4p | 0.002034-2p | 0.028670-2p |
| 7560-F | 1.161237-2p | 0.093498-8p | 1.160850-2p | 0.629270-16p |
| 7560-B | 1.161069-2p | 0.023938-2p | 0.006726-2p | 0.000317-2p |
| 7560-C | 0.091316-2p | 0.051727-2p | 0.022828-2p | 0.001772-2p |
| 7560-S | 0.034742-2p | 0.098711-2p | 0.002839-2p | 0.033446-2p |
| 15120-F | 1.487712-32p | 0.064557-8p | 1.388741-32p | 0.619617-16p |
| 15120-B | 1.540916-32p | 0.054789-8p | 0.473334-16p | 0.537965-24p |
| 15120-C | 1.297610-8p | 0.059854-8p | 0.713919-32p | 0.518768-16p |
| 15120-S | 0.847324-8p | 0.071499-8p | 0.661104-8p | 0.496004-16p |

Table 4.6: Best case C.O.V for automatic quadrature routine II

| | AFACT | WFACT | AWF |
|---|---|---|---|
| 3780-F | 0.061858-4p | 0.184691-8p | 0.166484-8p |
| 3780-B | 0.001456-2p | 0.038613-2p | 0.107041-2p |
| 3780-C | 0.060670-2p | 0.122445-2p | 0.093152-2p |
| 3780-S | 0.032092-2p | 0.014896-2p | 0.029798-2p |
| 7560-F | 0.063997-4p | 0.156076-16p | 0.155449-16p |
| 7560-B | 0.059428-2p | 0.018083-2p | 0.068948-2p |
| 7560-C | 0.055747-2p | 0.108365-2p | 0.096238-2p |
| 7560-S | 0.033780-2p | 0.033008-2p | 0.033341-2p |
| 15120-F | 0.156235-24p | 0.119569-16p | 0.108273-16p |
| 15120-B | 0.142170-8p | 0.168313-32p | 0.111841-24p |
| 15120-C | 0.162894-16p | 0.172315-24p | 0.129945-24p |
| 15120-S | 0.119453-8p | 0.156031-32p | 0.161307-16p |

Table 4.7: % cost improvement over static chunking for automatic quadrature routine

| | FSC | GSS | FACT | AFACT | WFACT | AWF |
|---|---|---|---|---|---|---|
| 3780-F | 69.49-8p | -0.1-2p | 43.65-8p | 68.25-8p | 67.02-8p | 69.78-8p |
| 3780-B | 72.03-8p | 62.79-16p | 63.73-4p | 72.95-8p | 72.89-8p | 69.18-8p |
| 3780-C | 68.64-16p | 40.07-32p | 51.63-16p | 65.80-16p | 66.76-32p | 67.51-16p |
| 3780-S | 59.42-16p | 23.31-32p | 43.86-32p | 62.46-16p | 60.88-16p | 60.94-16p |
| 7560-F | 71.97-8p | 2.83-32p | 47.87-16p | 71.70-8p | 72.53-16p | 73.12-8p |
| 7560-B | 74.37-8p | 62.66-16p | 65.94-4p | 73.40-8p | 75.27-16p | 75.37-16p |
| 7560-C | 70.64-16p | 40.96-32p | 52.22-16p | 70.27-16p | 72.08-16p | 72.71-16p |
| 7560-S | 69.45-16p | 21.80-16p | 44.32-32p | 69.20-16p | 63.13-32p | 69.44-16p |
| 15120-F | 74.96-8p | 5.84-32p | 49.21-16p | 72.42-8p | 73.99-16p | 73.02-16p |
| 15120-B | 76.47-16p | 63.99-16p | 52.23-8p | 74.42-8p | 70.53-32p | 72.59-16p |
| 15120-C | 74.38-16p | 41.72-32p | 53.01-16p | 72.23-32p | 69.02-32p | 68.98-24p |
| 15120-S | 73.36-24p | 22.93-16p | 51.55-24p | 70.68-24p | 65.02-32p | 68.41-16p |

The application suffers load imbalance because of the nature of distribution of the time consuming integrals. The four different types of load distribution used for the experiments are shown in the Figures 4.23 - 4.26. For front distribution, guided self scheduling performed as badly as static chunking for the same reason as before. Factoring performed marginally better than guided self scheduling also for the same reason as before. Since the optimal chunk size was empirically determined, fixed size chunking along with adaptive factoring gave the best performance for this distribution. In general, adaptive weighted factoring closely follows fixed size chunking and adaptive factoring and also marginally better than weighted factoring. On an average, factoring gave up to 45% cost improvement over static chunking whereas weighted factoring and adaptive weighted factoring gave up to 73%. Also, fixed size chunking and adaptive factoring gave up to 74% and 72% cost improvement over static chunking respectively.

For back distribution, all the techniques consistently outperformed static chunking. Factoring performed poor for the same reason. Since, in back distribution, major work is distributed at the back, guided self scheduling performed better than factoring. Again, fixed size chunking and adaptive factoring were the best performing techniques. The weighted variants of factoring (weighted factoring and adaptive weighted factoring) significantly outperformed factoring. With the scaling up of problem size, adaptive weighted factoring performed better than the weighted factoring. The highest cost improvement (76%) was obtained for 15120 problem size by fixed size

chunking. Adaptive factoring gave up to 74% cost improvement. Weighted factoring and adaptive weighted factoring gave over 70% cost improvement. Guided self scheduling and factoring gave up to 63% cost improvement over static chunking.

For center distribution, all the techniques consistently outperformed static chunking. Factoring performed better than guided self scheduling. Adaptive weighted factoring and weighted factoring consistently outscored factoring. Adaptive weighted factoring performed as good as weighted factoring in most cases and in some cases performed better than weighted factoring. Adaptive factoring and fixed size chunking were the two best performing technique. Here too, the cost improvement factor for fixed size chunking, adaptive factoring, weighted factoring, and adaptive weighted factoring was over 70%. For factoring and guided self scheduling, that mark was around 50% and 40% respectively.

In scatter distribution, load varies continuously in the manner shown in the Figure 4.26. Fixed size chunking and adaptive factoring performed best and guided self scheduling performed worst. Adaptive weighted factoring closely follows adaptive factoring. Both weighted factoring and adaptive weighted factoring consistently outperformed factoring. Since, in this distribution load varies continuously, adaptive weighted factoring was able to adapt to the changing environment and thus gave better performance than weighted factoring. In general, with increasing problem size, all the techniques gave increasing cost improvement. The two best cost improvement was little over 73% by fixed size chunking and 70.68% by adaptive factoring. Table 4.5 and 4.6 shows the best case coefficient of variation (c.o.v) of processor finishing times of various techniques. The c.o.v of all the techniques are consistently lower than the c.o.v of static chunking which underscores the importance of the techniques.

## 4.7 Heat Solver

This application is an unstructured grid heat solver. The solver computes the solution to the steady state heat conduction equation (Laplace equation) using Jacobi method and it proceeds over a number of time-steps. During each time step, the solver sweeps over the computational domain to compute the temperature norm by solving the heat conduction equation on all grid points. The time-stepping stops when the temperature norm is within the acceptable tolerance level. The solver reads a grid file as an input file which is a triangulated grid. The solver

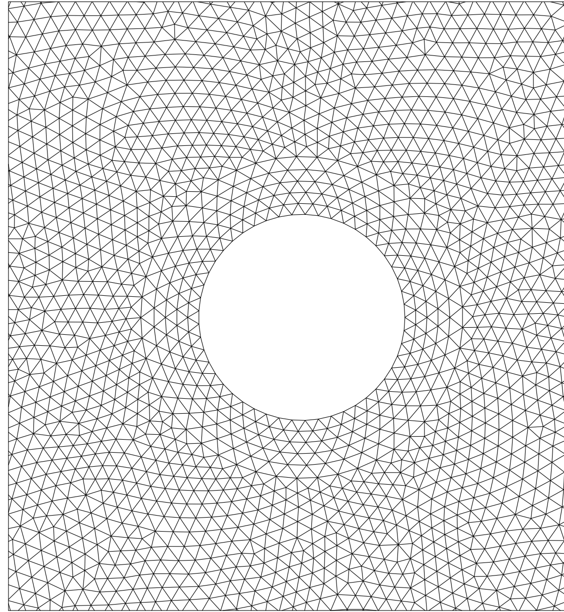

Figure 4.51: Cylinder

also reads other parameters like the coordinates of the elements, initial temperature on each nodes, boundary normal, element connectivity which is a three point number for each triangular element, etc. The problem can be scaled by increasing the size of the input grid. Testing was done on a cylindrical test case with 105k, 148k, 200k and 300k grid points. The Figure 4.51 shows unstructured mesh over a cylinder.

Table 4.8: % cost improvement over static chunking for heat solver

| | FSC | GSS | FACT | AFACT | WFACT | AWF |
|---|---|---|---|---|---|---|
| 105k | 30.23-8p | 39.76-4p | 28.45-8p | 28.06-8p | 35.00-8p | 34.47-4p |
| 148k | 40.62-8p | 47.44-8p | 36.80-8p | 38.99-8p | 47.57-8p | 46.25-8p |
| 200k | 48.78-8p | 53.25-8p | 38.41-8p | 44.10-8p | 50.60-8p | 50.40-8p |
| 300k | 53.72-8p | 55.72-16p | 40.43-8p | 50.07-8p | 52.26-8p | 52.55-8p |

The experiments were conducted on four different grid sizes. All the techniques performed better than static chunking. However, there is no significant difference in performance between

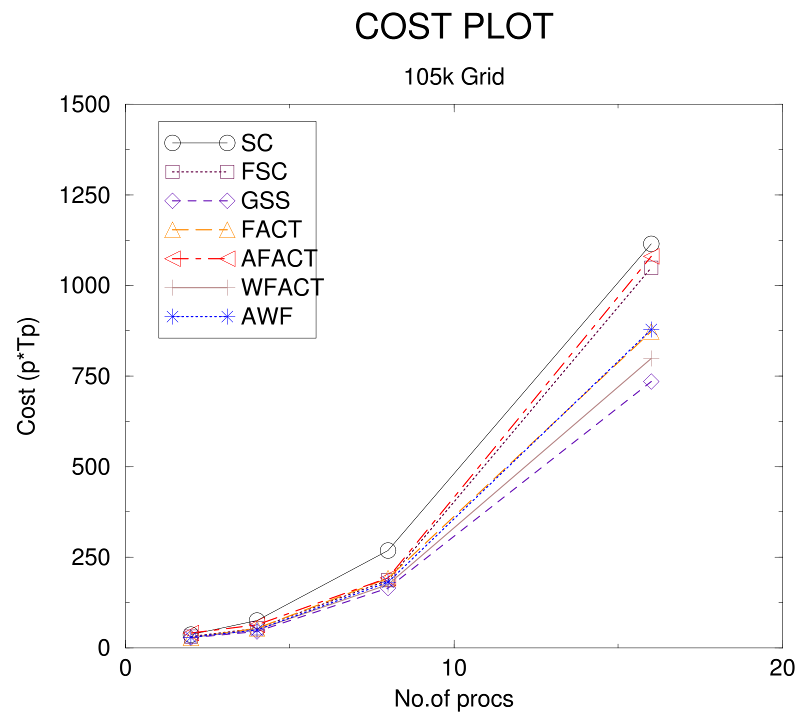


Figure 4.52: Cost: 105K grid

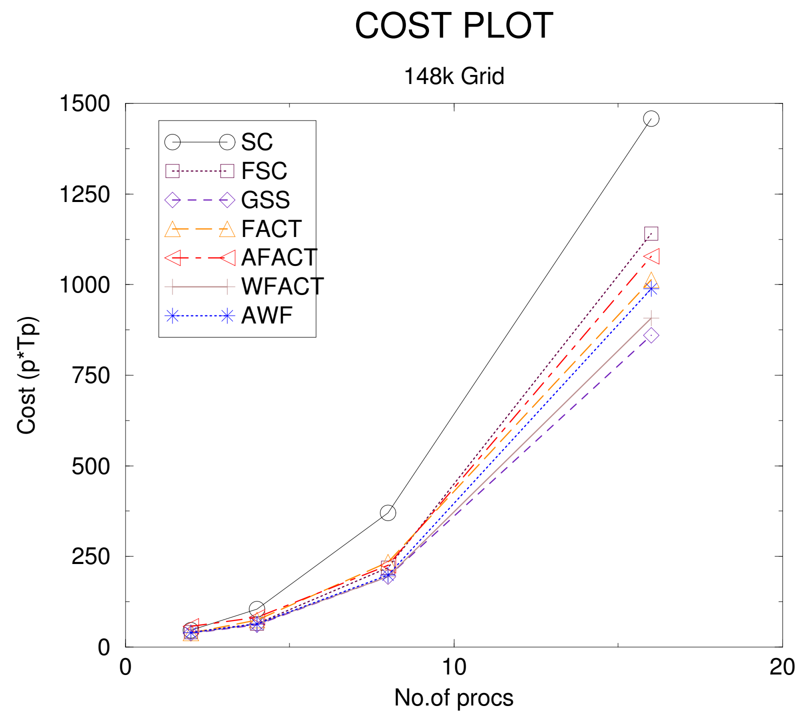


Figure 4.53: Cost: 148K grid

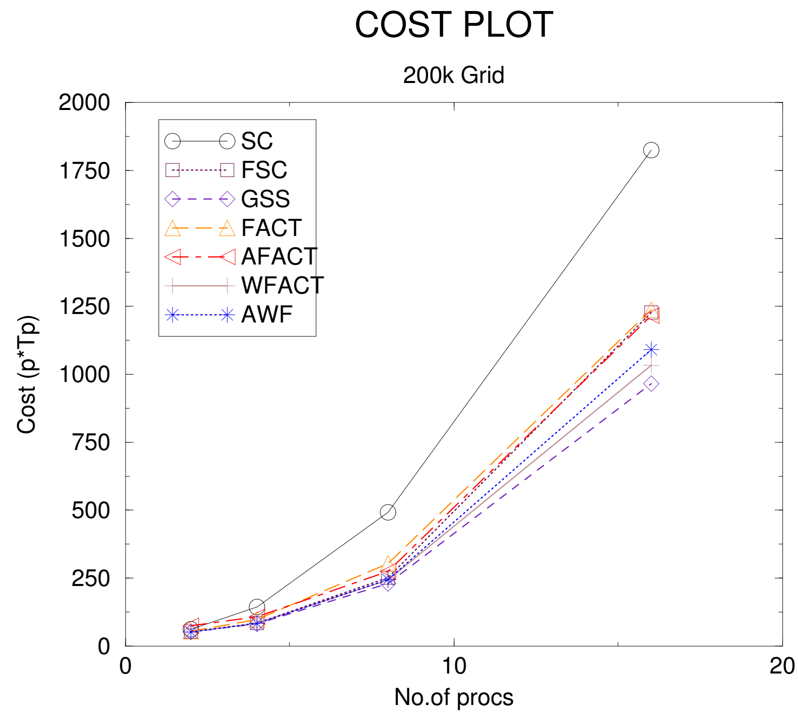


Figure 4.54: Cost: 200K grid

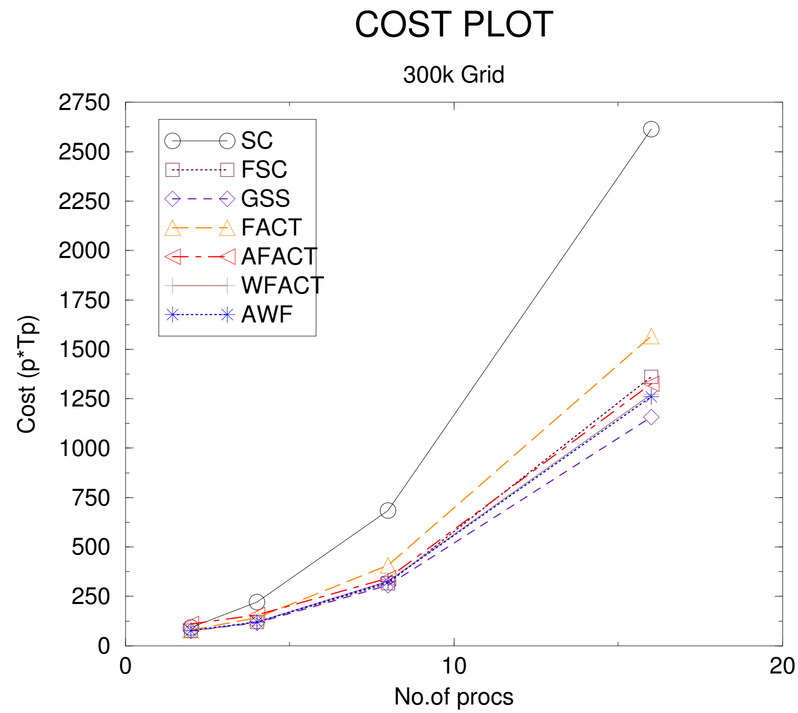


Figure 4.55: Cost: 300K grid

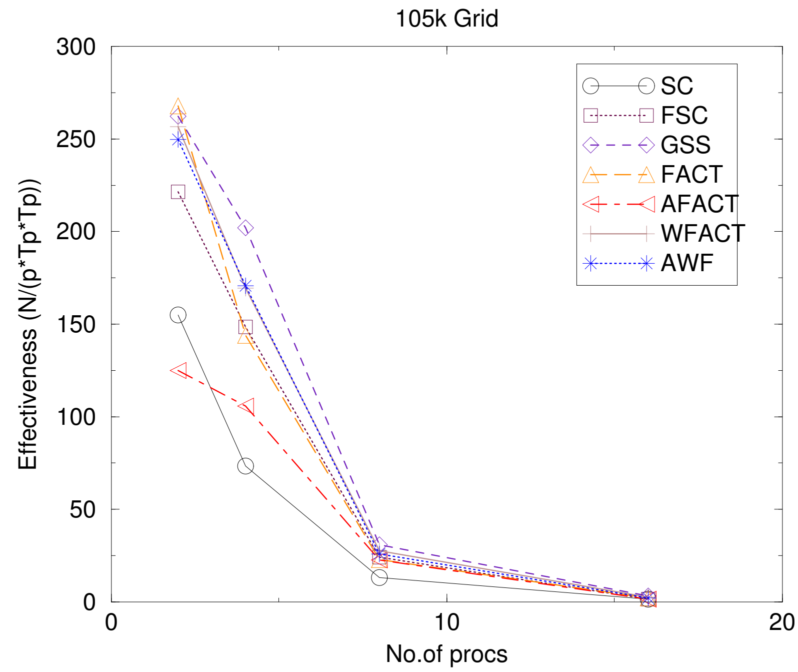


Figure 4.56: Effectiveness: 105K grid

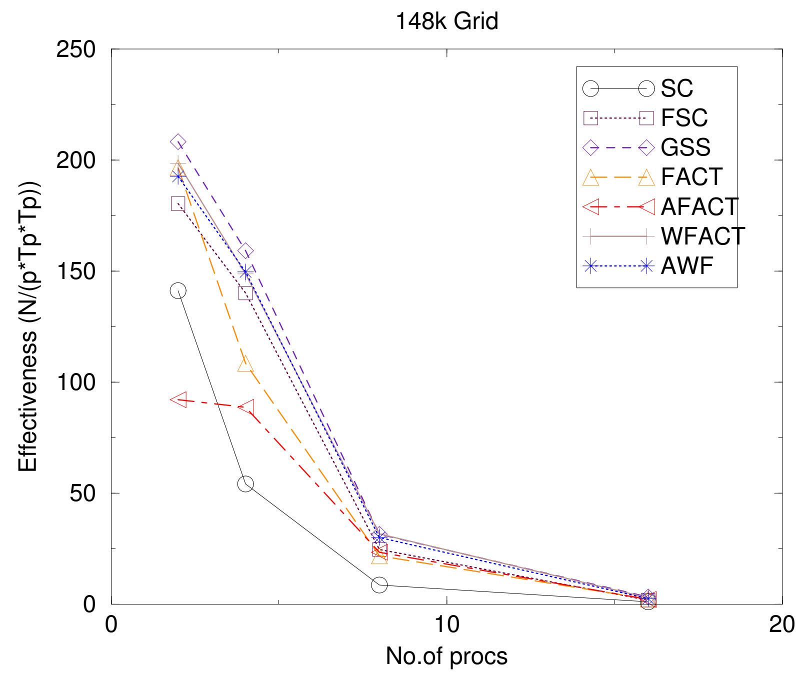


Figure 4.57: Effectiveness: 148K grid

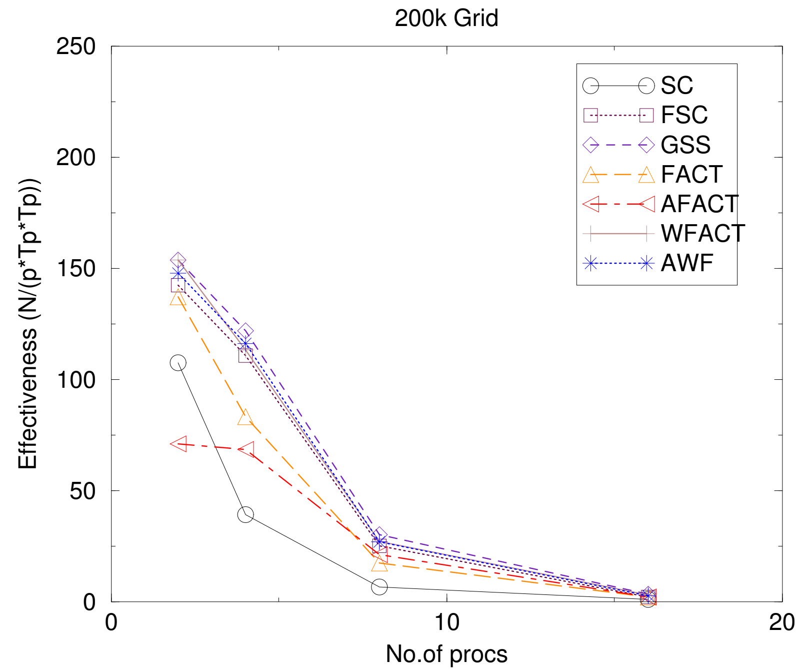


Figure 4.58: Effectiveness: 200K grid

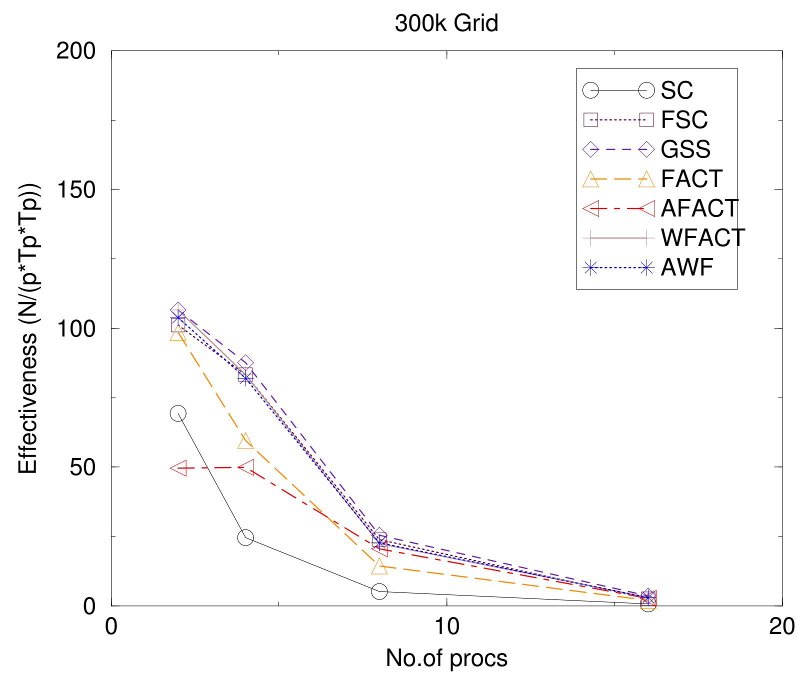


Figure 4.59: Effectiveness: 300K grid

the techniques. Even as the input grid size is increased, the performance of the techniques follows a similar pattern. The graphs suggest that guided self scheduling performed marginally better than most other techniques. Factoring and weighted versions of factoring (weighted factoring and adaptive weighted factoring) performed almost similar . This can be attributed to the absence of load variation (at least from the problem) as the computation proceeds. Fixed size chunking does not seem to perform as well as it performed with the other two application because of non-optimal chunk size. Adaptive factoring performed better than few techniques but was not the best performing technique because of the large ratio of the scheduling overhead to the computational time. The cost improvement values are presented in Table 4.7.

## 4.8 Summary

In this chapter, a detailed description of the test cases considered along with the performance metrics that were used to evaluate the scheduling techniques are presented. The three applications considered are the $N$-body simulations, automatic quadrature routine and an unstructured grid heat solver. The results of the conducted experiments were presented and interpreted. Based on the experiments conducted, the following things can be concluded: Fixed size chunking performs very well if optimal chunk-size can be found. However, finding the optimal chunk-size may require a number of trial and error observation. Adaptive factoring seems to work well for certain type of problem characteristics. For example, in $N$-body simulations, it worked very well for Gaussian distribution and in automatic quadrature routine, it worked well for all types of distribution, in particular, the scatter distribution where the load variation is highly irregular. This shows that adaptive factoring is indeed the choice of scheduling technique for environments characterized by highly irregular behavior. However, for environments where the scheduling overhead is dominant (like in the heat solver) adaptive factoring may not give best performance. Factoring seems to suffer if the majority of the load is scheduled during the first chunk. For problems with this characteristic, factoring does not gives good performance. However, if majority of the work is at the back then it favors factoring. The weighted variants of factoring (weighted factoring and adaptive weighted factoring) in general performed better than factoring. However, their performance depends on the initial chunk-size to determine the weights of the processors. If the initial chunk-size is not properly determined, then they may not give better performance

than factoring. Determining initial chunk-size is entirely application specific. So, the library provides the functionalities to set the initial chunk-size. Therefore, it is up to the user who has knowledge about the application to set the initial chunk-size. Adaptive weighted factoring performed better than weighted factoring when the load variation is not uniform. This is because adaptive weighted factoring recalculates the weights of the processors after every batch. Guided self scheduling also seem to suffer from the same problem afflicting the factoring. Hence, for problems where major portion of the work is at the front, guided self scheduling may not give good performance. However, if the major portion of the work is present at the back, then it is expected to give good performance.

Different techniques performed very well under different running conditions. In fact, this was the motivation behind developing such a library package which includes all the loop scheduling techniques. So, it is up to the user who has knowledge about the application to select the scheduling technique according to the application characteristics and the running conditions. The explanation given in the previous paragraph will assist the application developer in choosing a particular technique over another technique for the application.

# CHAPTER V

# OVERHEAD ANALYSIS

This chapter attempts to evaluate the overhead of the runtime system *parallel runtime environment for multicomputer architecture* (PREMA) vis-a-vis the native message layer MPI. For this purpose, two different sets of experiments were conducted. The first set of experiments consist of latency and startup time measurements and the next set of experiments was related to the performance of the runtime system and the messaging layer with respect to the test applications.

This chapter is organized as follows: In section 5.1, the latency measurements and the startup time measurements are given. The experimental evaluation is presented in section 5.2. An analytical evaluation of the experimental results is presented in section 5.3.

## 5.1 Latency Measurements

The presence of software in the critical path between the host system memory and the network interface worsens the raw latency of the network system. The performance of the network system can be evaluated by using a test program that transfers messages of fixed size across two nodes. The latency is then measured as the time required to transfer a message from a sender to a receiver and the time required to inform the sender of the completion of the message transfer. Since PREMA is a software that runs on top of the messaging software (MPI), it is expected that the latency of the PREMA system would be greater than the latency of the messaging layer. The UltraMSPARC cluster used for these experiments is a 16 node cluster. Each node has 4 400 MHz UltraSPARC II processors. The nodes are connected through a Myrinet switch as well as 100 Mb/s Ethernet. The MPICH implementation of MPI is used for the experiments.

The code snippet for PREMA and MPI are shown in the Figure 5.1 and Figure 5.2 respectively. The Figure 5.1 also illustrates the way PREMA is used: the receiving side must explicitly poll

```
int flag;
void requestHandler(int src, void *data, int size, void *arg) {
    if (myId) {
        mol_request((myId+1)%num_procs,........);
    }
    flag++
}

/*sender*/
start = gethrtime();
for (i=0;i<repeat;i++) {
    mol_request(1,......);
    while (!flag)
        mol_poll();
    flag=0;
}
end = gethrtime();
printf("Latency:  %lf\n",(end-start)/2);

/*receiver*/
for (;;) mol_poll();
```

Figure 5.1: Latency measurement using polls and requests

```
/*sender*/
start = gethrtime();
for (i=0;i<repeat;i++) {
    MPI_Send(...);
    MPI_Recv(...);
}
end = gethrtime();
printf("Latency: %lf\n",(end-start)/2);

/*receiver*/
for (;;) {
    MPI_Recv(...);
    MPI_Send(...);
}
```

Figure 5.2: Latency measurement using sends and receives

as shown in the figure and the separation of the communication and the synchronization events. The handler used in *mol_request* will be one among the five handlers described in chapter 3. Generally, the handler will incur some work before requesting the master for additional work. In order to model the work time, a small amount of wait time was introduced at the receiver side. The Figure 5.4 and Figure 5.3 represents the plot between the transfer time and the message size with and without the wait time respectively. The experiments were conducted between two processors of the same node as well as between two processors of different nodes. The Figure 5.3 depicts the performance of the runtime system and the messaging layer without the wait time. From the plot, it is evident that the inter-node latency is almost ten times higher than the intra-node latency. Up to 10kb message size, MPI completely outperforms PREMA for both inter-node as well as intra-node. However, with increasing message sizes, the difference diminishes. This shows, on an average, PREMA overhead is a couple of magnitude larger than the messaging layer overhead. With the introduction of 0.25 seconds wait time at the receiver side, the transfer times of both PREMA and MPI are almost similar as shown in Figure 5.4. There seem to be little difference between inter-node and intra-node latency. This is because, the major factor that dominates the latency is the wait time. These two figures reveal that the handler execution (RSR) is a bit costlier operation when the amount of work incurred during the execution of the handler is very small when compared to the cost of executing the handler. Since, RSRs are not supported by the messaging layer, the extra cost has to be paid.

The Figure 5.5 represents the plot between the startup time against different message sizes. Startup time refers to the time difference between two successive calls to the runtime library. The results reported are for 25 consecutive function calls. However, in real applications, the number of times the handler is executed will be in the order of 100s or 1000s depending upon the problem size and the number of processor used to solve the problem. This plot is almost similar to Figure 5.3 i.e. for small message sizes, MPI completely outperforms PREMA and for larger message sizes, the difference diminish. The startup time for the runtime system is consistently higher than the startup time of the messaging layer for all message sizes. This difference can be very substantial when the number of scheduling operations is very large. The startup time coupled together with the handler execution cost can be significant under certain conditions.

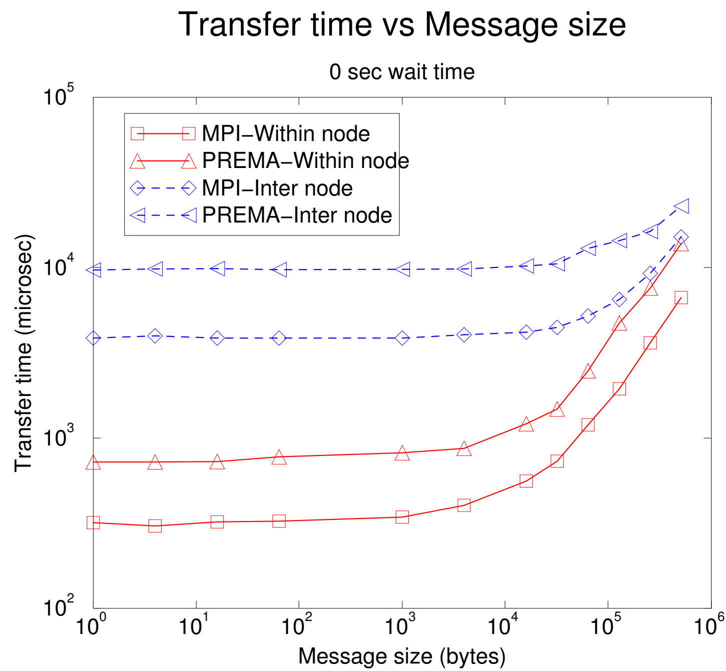


Figure 5.3: Transfer time vs Message size with 0 sec wait time

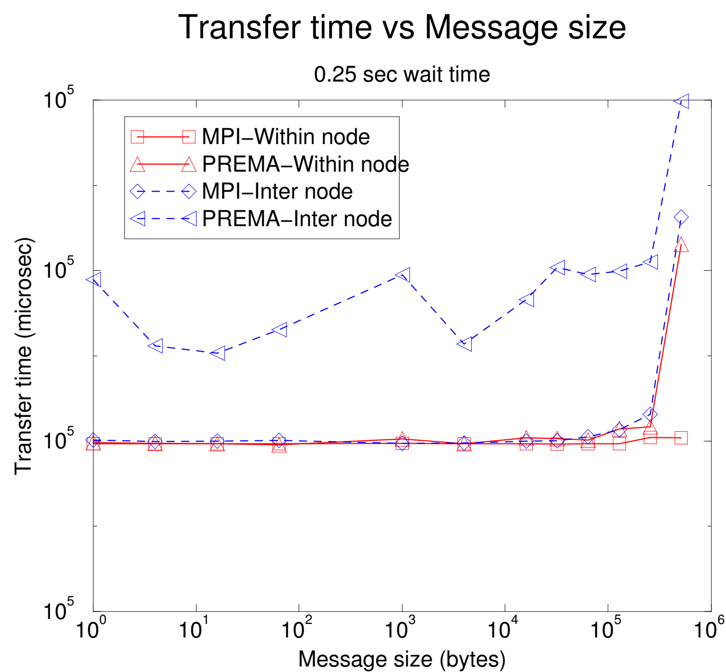


Figure 5.4: Transfer time vs Message size with 0.25 sec wait time

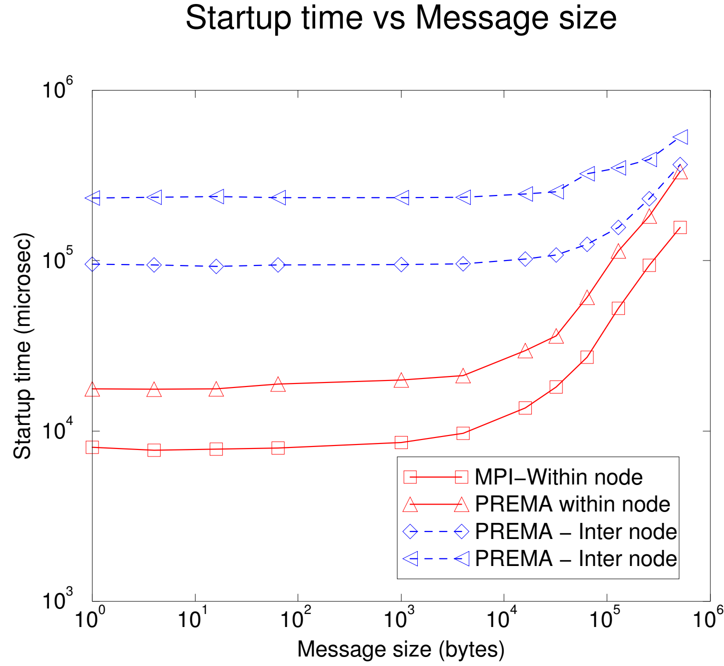


Figure 5.5: Startup time vs Message size

### 5.2 Experimental Evaluation

Having seen the latency and the startup cost of the runtime library against the messaging layer in the previous section, in this section we shall see how these factors influence the performance of real applications. For this purpose, two different implementations of the same scheduling technique, one using the runtime system and another using the messaging layer are compared. The runtime system overhead is incurred during every scheduling operation. For techniques like fixed size chunking, guided self scheduling, and factoring, the number of scheduling operation ($N_{sch}$) can be determined before runtime provided the number of loop iterates and the number of processors that is used to solve the problem are known. However, for adaptive factoring technique, calculating chunk size involves some parameters that are determined entirely at the runtime. Therefore, it is not possible to know $N_{sch}$ before hand. For weighted factoring and adaptive weighted factoring, though $N_{sch}$ is same as factoring, the exact processor running conditions cannot be replicated. Therefore, for adaptive factoring, weighted factoring and adaptive weighted factoring, the results would not be a direct indicative of the overhead introduced by PREMA. For this reason, the comparative results are presented only for fixed size chunking, guided self

scheduling and factoring. Since, any form of overhead results in increased cost, *cost* performance metric is used for comparison purpose. It should be mentioned that the extra overhead suffered by the application because of the runtime system also leads to load imbalance. This in turn, can affect the task allocation to individual processors. Therefore, it is outrightly not possible to predict whether MPI implementation will always be better than the PREMA implementation. Since, during runtime, the assignment of work to processors is non-deterministic, it is entirely not possible to single out the contribution of the runtime system to the performance difference. Nonetheless, the cost comparison between two different implementation would roughly give us an idea of the performance degradation/improvement suffered by the application. The % cost variation was calculated as follows:

$$\%\text{ cost variation} = \frac{(cost_{prema} - cost_{mpi}) * 100}{cost_{mpi}}$$

### 5.2.1 $N$-body Simulations

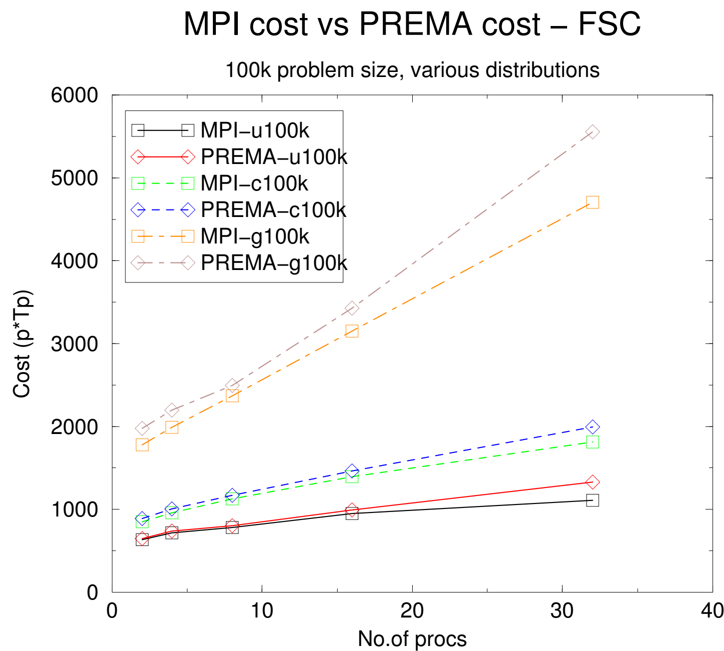


Figure 5.6: $N$-body-FSC cost comparison

The figures in this subsection represents the cost plot and the % cost variation plot for the two different implementation of the scheduling techniques. From Figures 5.6 - 5.8, reveal that PREMA implementation of the scheduling technique closely follows the MPI implementation of

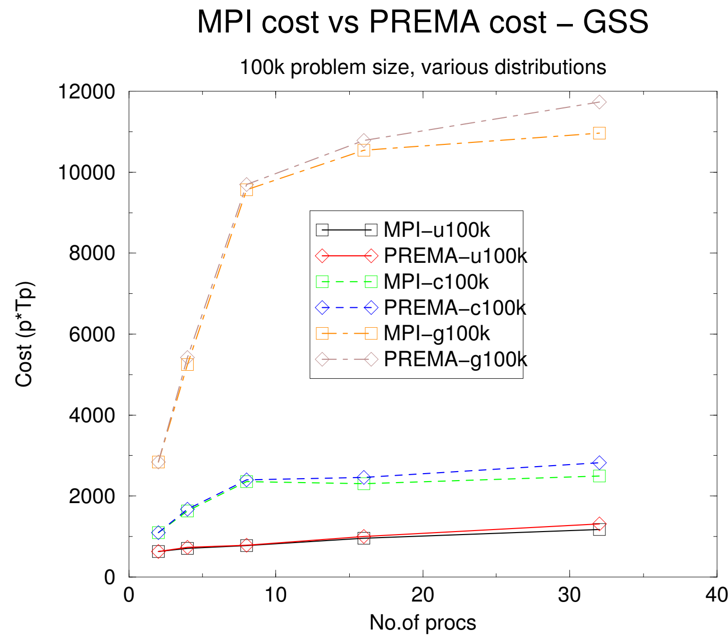


Figure 5.7: $N$-body-GSS cost comparison

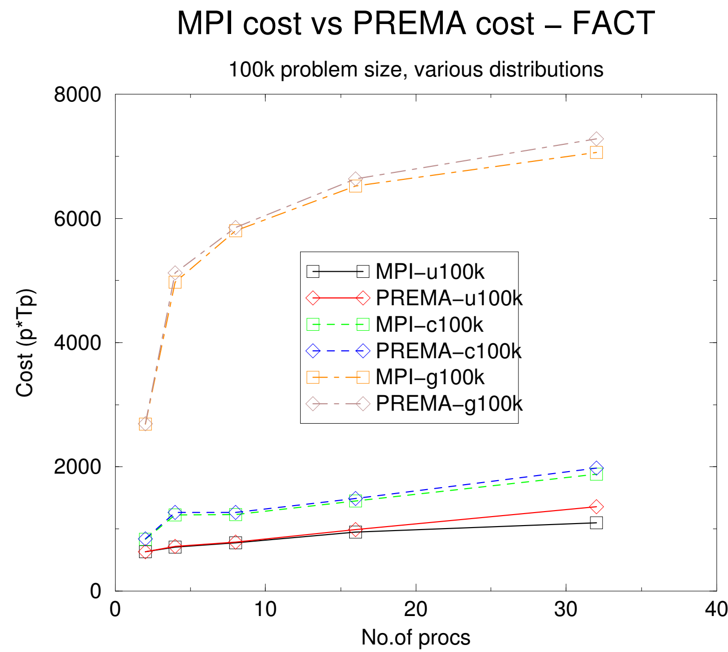


Figure 5.8: $N$-body-FACT cost comparison

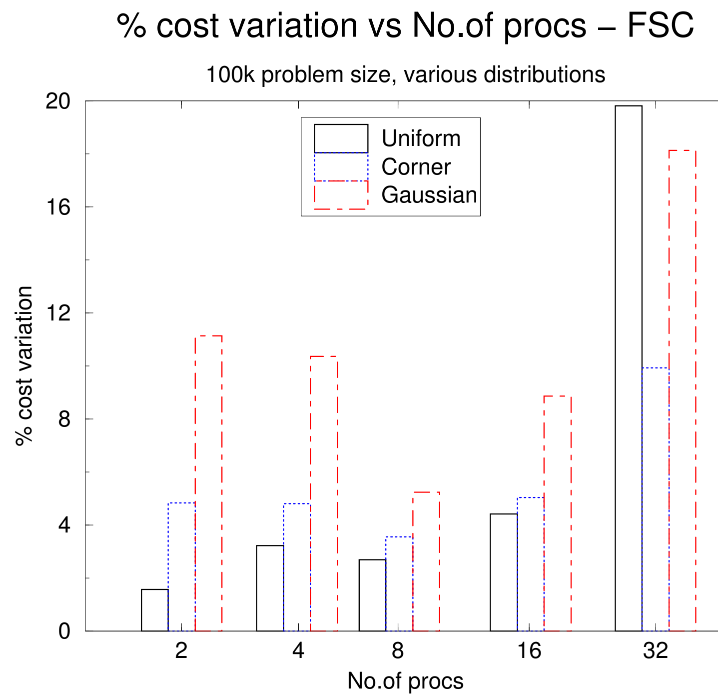


Figure 5.9: % cost variation of $N$-body-FSC

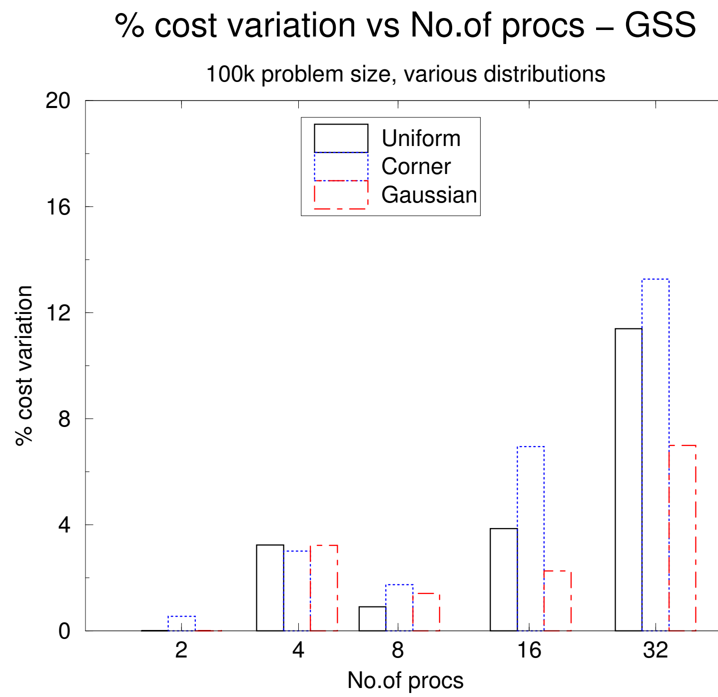


Figure 5.10: % cost variation $N$-body-GSS

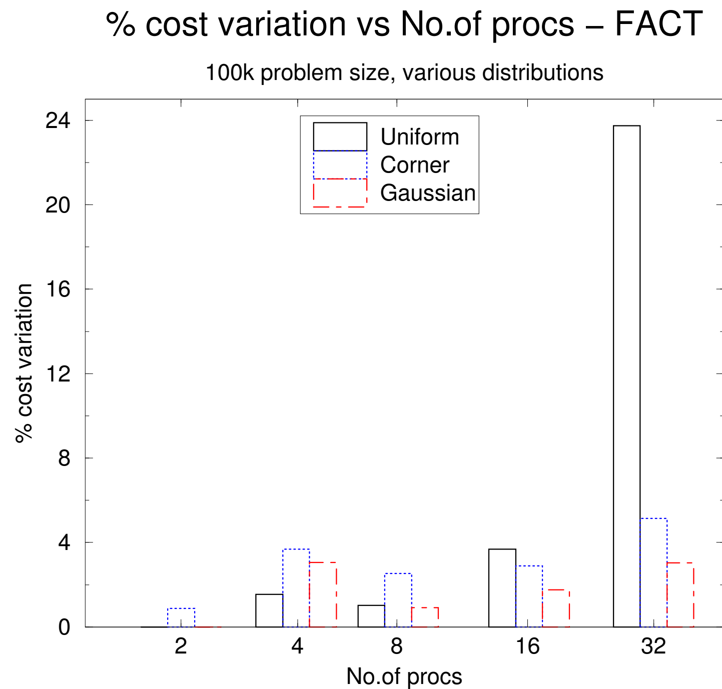


Figure 5.11: % cost variation $N$-body-FACT

the same scheduling technique. Whatever little difference is present occurs when the number of processors used is high. In this case, it is 32. The cost plot gives us just the qualitative difference between the two implementations. In order to understand the quantitative difference, a bar chart which gives the % cost variation for the two different implementations is given in Figures 5.9 - 5.11. These figures also show how different scheduling technique accommodate the overhead of PREMA. That's the reason why the figures are different for different techniques. For factoring, except for uniform distribution at 32 processors, the overhead is below 5%. For guided self scheduling up to 16 processors, on an average, the overhead is below 5% and for fixed size chunking up to 16 processors, on an average, the overhead is between 5%-10%. The worst case for fixed size chunking occurs at 32 processors where for all distributions, the overhead is above 10% with the maximum at just below 20% for uniform distribution. However, for guided self scheduling, that mark is around 12% for uniform and corner distribution and below 8% for Gaussian distribution. For factoring, the worst case occurs at 32 processors for uniform distributions at just below 24% (highest among all comparisons). Since, guided self scheduling and factoring are decreasing size chunk schemes, they adapt better than the fixed size chunking.

In general, the % cost variation is higher at 32 processors than at any other number of processors below that. From the last section we saw that the startup time of the PREMA functional cost is high and the handler execution cost is also high when the amount of work performed in the handler is low. Since, for a given problem size, $N_{sch}$ is at highest for 32 processors, and also the amount of work to be done by each processor is at the lowest, it is possible that high startup cost and the high handler execution cost contributes to this observation. The nature of the work distribution along with the scheduling technique also plays a role here. That's the reason, the % cost variation is different for same distribution at the same number of processors for different scheduling techniques.

### 5.2.2 Automatic Quadrature Routine

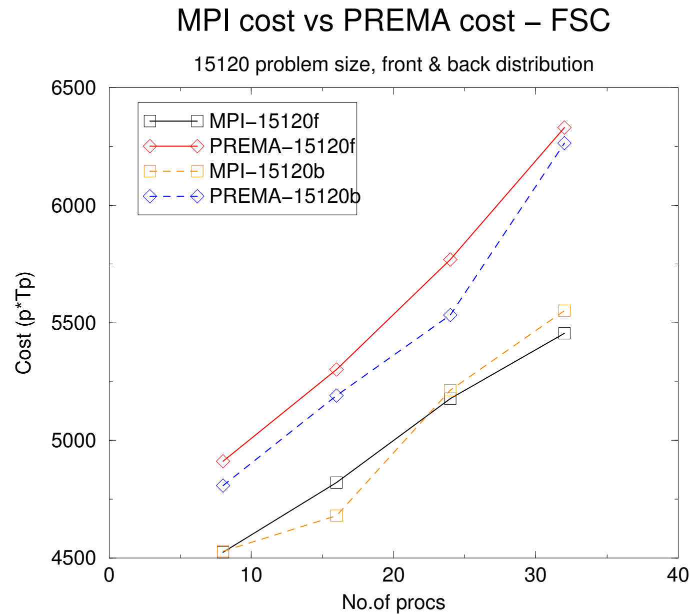


Figure 5.12: AQR FSC cost comparison I

The Figures 5.12 - 5.17, represents the cost comparison plot for fixed size chunking, guided self scheduling and factoring. From the plots, we see that both the implementations closely follow each other. The next three Figures 5.18 - 5.20, reveals an interesting observation. For center and scatter distribution, the PREMA implementation of guided self scheduling and factoring is actually better than the MPI implementation. As we have seen before, it is possible that the overhead of the runtime system affects the scheduling decision. In this case, it has affected the

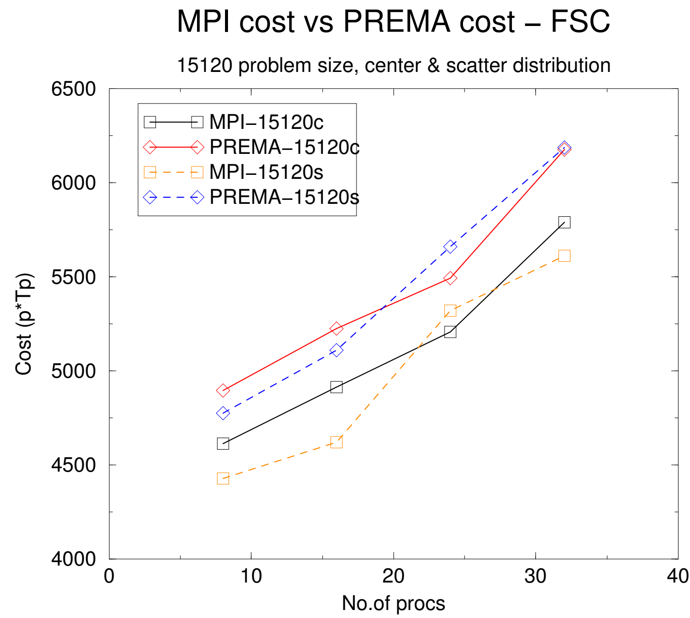


Figure 5.13: AQR FSC cost comparison II

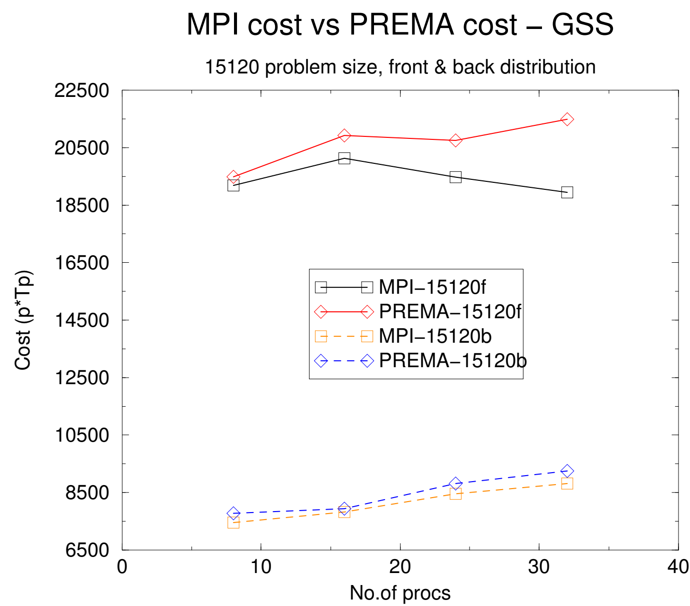


Figure 5.14: AQR GSS cost comparison I

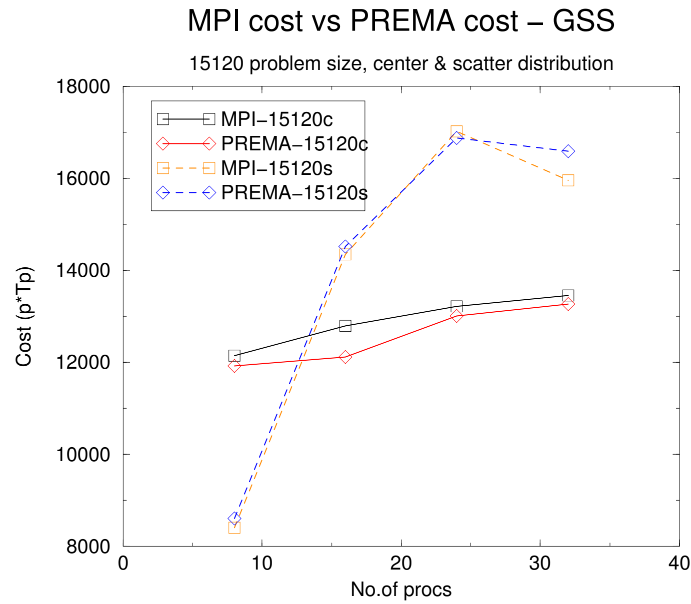


Figure 5.15: AQR GSS cost comparison II

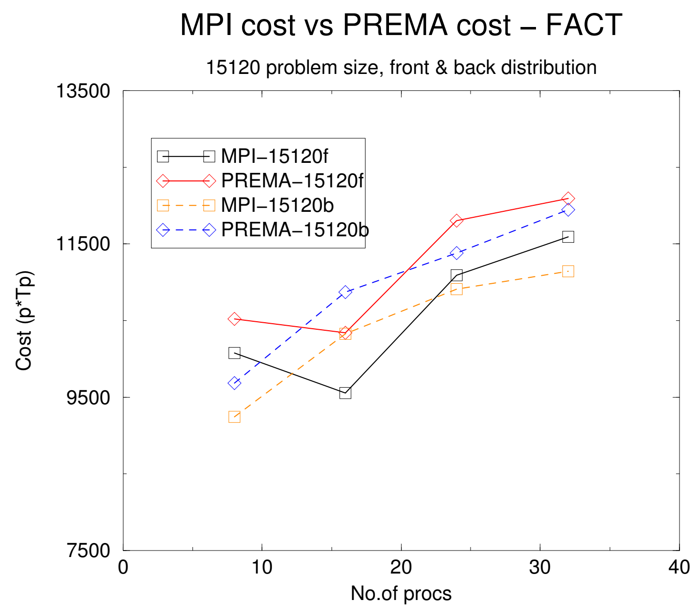


Figure 5.16: AQR FACT cost comparison I

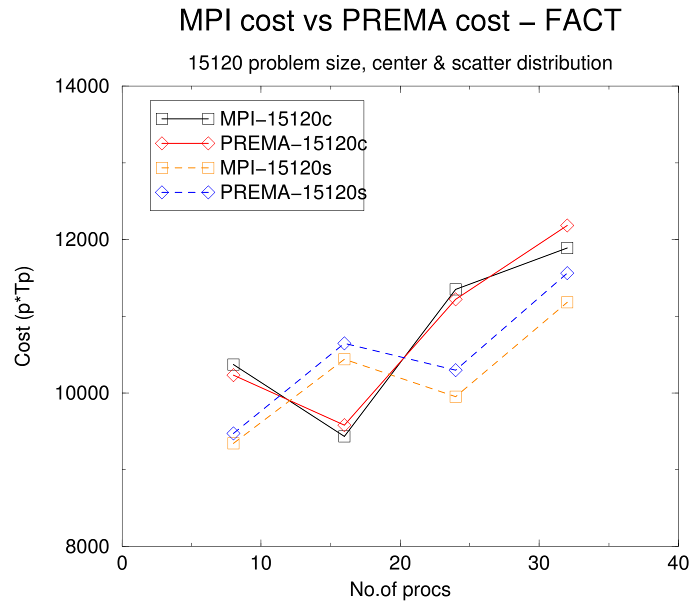


Figure 5.17: AQR FACT cost comparison II

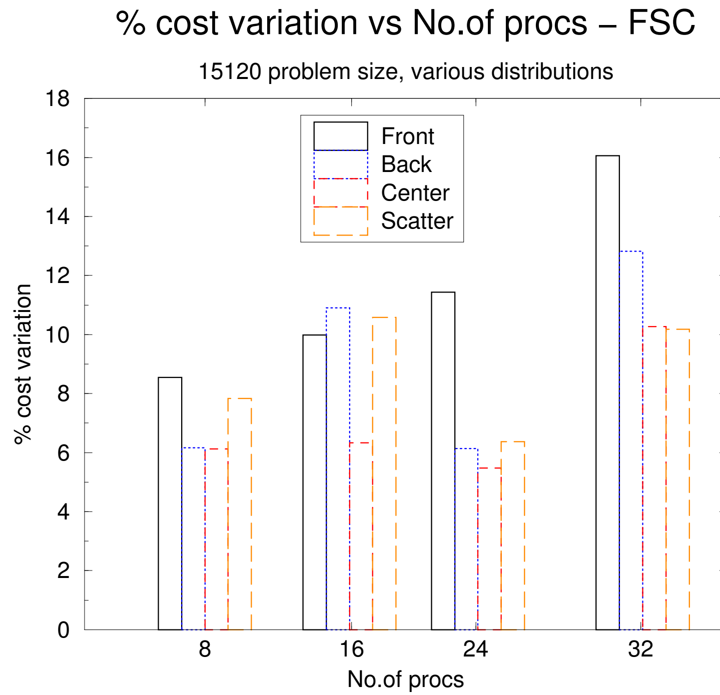


Figure 5.18: % cost variation of AQR-FSC

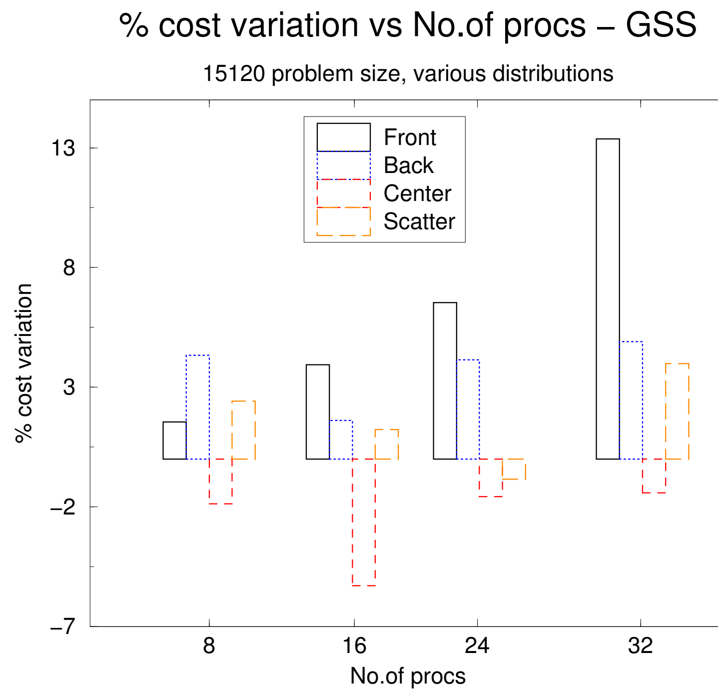


Figure 5.19: % cost variation AQR-GSS

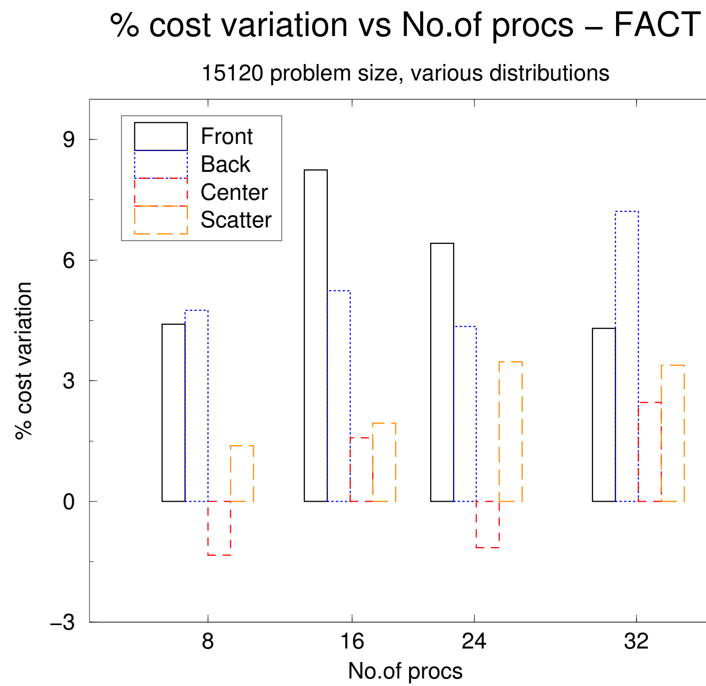


Figure 5.20: % cost variation AQR-FACT

load distribution among the processors. That's the reason why PREMA implementation was slightly better than the MPI implementation. For factoring, the maximum cost increase (8%) occurs for front distribution and for all other distributions, that figure is below (7%). In guided self scheduling, the front distribution suffers the maximum cost increase (13%) at 32 processors. Apart from this, for all other distributions, the % cost increase is below 5.5%. Fixed size chunking suffers maximum % cost increase when compared to other two techniques. The front distribution suffers 16% cost increase at 32 processors. Other distributions suffer 6-12% cost increase for different number of processors.

### 5.2.3 Heat Solver

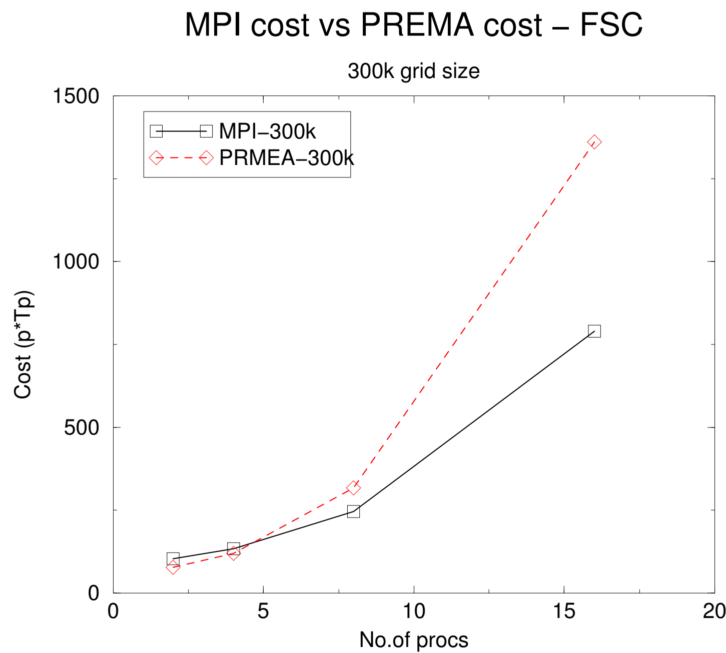


Figure 5.21: Heat Solver FSC cost comparison

This application is not very large and there is very little load imbalance in the problem. From the Figures 5.21 - 5.24, it is evident that there is a substantial cost difference when the number of processors is increased beyond four. The % cost variation plot also does not show any consistent observation. For two and four processors, the PREMA implementation is marginally better than the MPI implementation. But at eight and sixteen processors, the % cost increase is very high. Since the application is really small with little load imbalance, it suffered most from

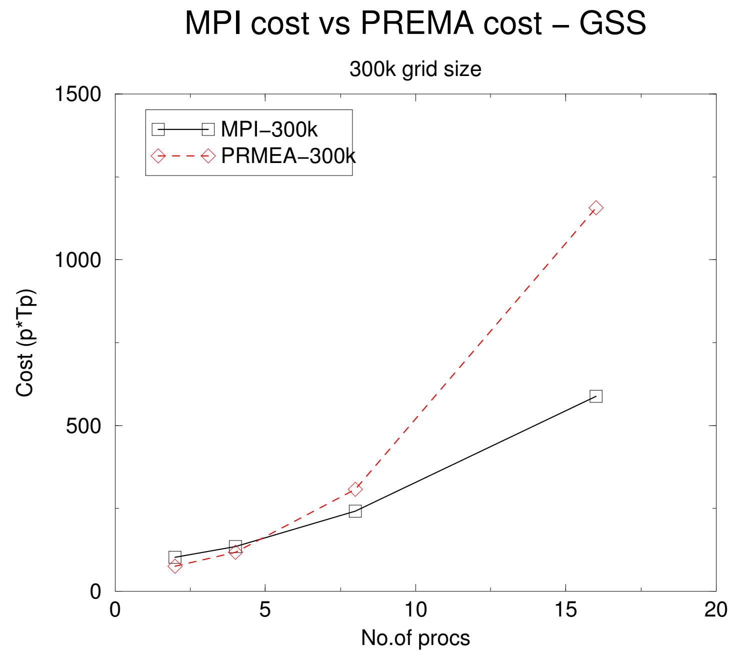


Figure 5.22: Heat Solver GSS cost comparison

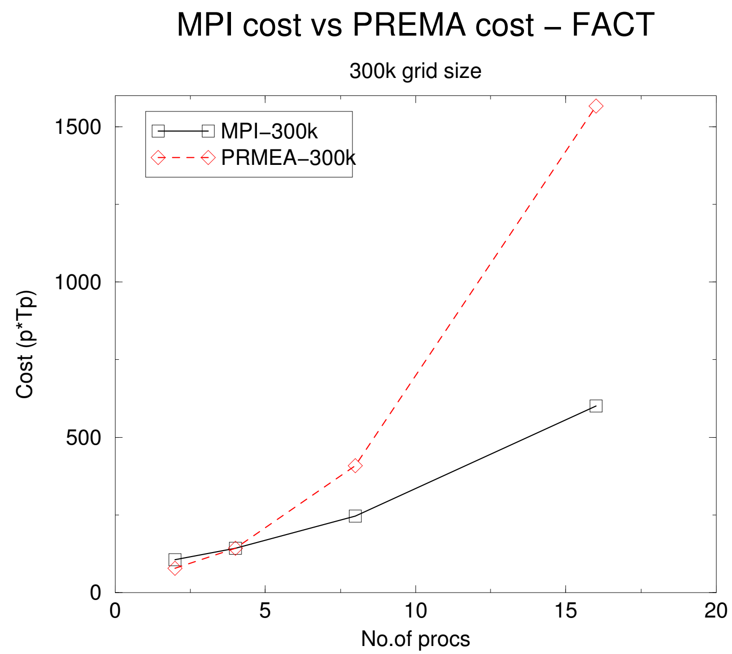


Figure 5.23: Heat Solver FACT cost comparison

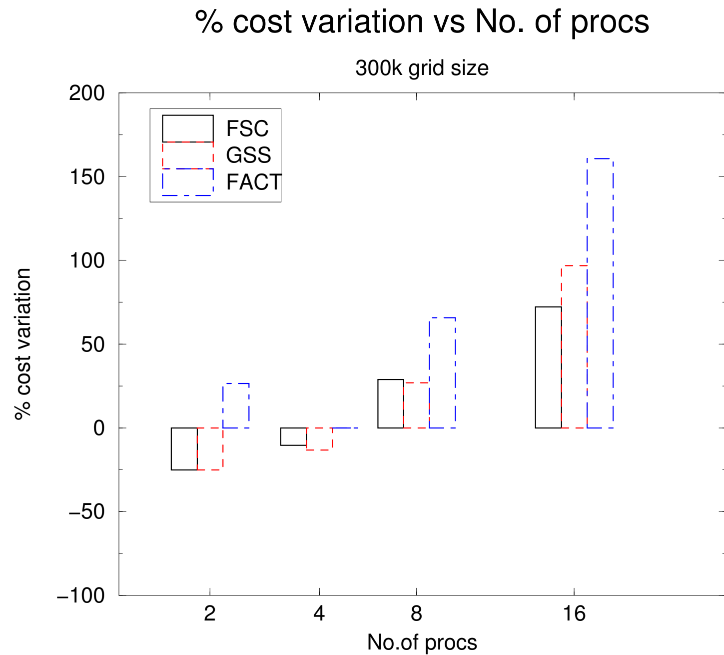


Figure 5.24: % cost variation for heat solver

the weakness of the runtime system: high latency and high startup cost. From the first section of this chapter we saw that the handler execution is a costly operation especially when the amount of work involved with the handler is small when compared to the overhead. By keeping the problem size fixed and increasing the number of processors, the amount of work to be performed by each processor is reduced. Since the overall problem size is itself small, this leads to increased overhead i.e., time spent in performing useful work is not proportionate to the amount of work done because of high latency and high startup cost of the runtime system.

## 5.3 Analytical Evaluation

This section presents the analytical evaluation that support the experimental results presented in the previous section. In parallel computing, all the processors cooperate to solve the problem by alternating between the computation and the communication phase. The computation time is the useful time spent by the processor to solve the problem and the communication time is the time spent by the processor for communication which is a source of overhead. In the master-slave approach, the communication time includes the interval of time between the moment a processor

requests work and the moment it receives work from the master, the time spent by the processor to pack the data and send it to another processor during data redistribution, etc. In addition to these forms of overhead, the master also suffers additional overhead because of the time required to do scheduling operation.

Let $O_{ML}$ be the overhead of the underlying messaging layer and $O_{RTS}$ be the overhead of the runtime system. For the purpose of simplicity, we can assume $O_{RTS} = xO_{ML}$ where $x$ is some factor. Therefore, the overhead suffered by the application because of the communication substrate is given by $O_{ML}+xO_{ML} = (1+x)O_{ML}$ and the additional overhead suffered because of the runtime system is given by $xO_{ML}$. This is the extra overhead suffered by the application for every scheduling operation. If this overhead is suffered by the application processes at the same instance of time for every scheduling operation, then the extra cost suffered by the application because of the runtime system would be proportional to this overhead. However, in reality, it is possible that the application processes suffer this overhead at different instance of time. Therefore, it is extremely difficult to estimate the overhead suffered by the application because of the runtime system.

Let us consider two scenarios. In scenario I, assume there are $N$ iterates and $P$ processors. Let the scheduling technique be *fixed size chunking* with $\frac{N}{2P}$ as the chunk size. Also assume that all the iterates take fixed amount of time to complete, say $x$ units of time. Let $t_{wait}$ be the time instance between when the processor requests for work and the master replies back with the work. The total number of scheduling operations with $\frac{N}{2P}$ as the chunk size is $2P$. Since, all the iterates take same amount of time to solve, it is safe to assume that each processor does $\frac{N}{P}$ iterates which involves $\frac{\frac{N}{P}}{\frac{N}{2P}} = 2$ scheduling operations. Therefore, the application cost is given by $P * (\frac{Nx}{P} + 2t_{wait})$.

In scenario II, let us assume there are $N$ iterates and $P$ processors, with fixed size chunking as the scheduling technique with $\frac{N}{2P}$ as the chunk size. Also assume that in this scenario the iterates have variable execution time and it takes $2x$ time units to solve each of the first $\frac{N}{2P}$ iterates and $0.5x$ time units to solve each of the rest $(2P-1) * \frac{N}{2P}$ iterates. Therefore, in this scenario, the processor which gets the first $\frac{N}{2P}$ iterates will consume $\frac{N}{2P} * 2x = \frac{Nx}{P}$ time units and each of the other processors will consume $\frac{N}{P} * 0.5x = \frac{Nx}{2P}$ time units. One among these processors will also consume extra $\frac{N}{2P} * 0.5x = \frac{Nx}{4P}$ time units. Therefore, the total running time of the application will be the $max(\frac{Nx}{P} + t_{wait}, \frac{3Nx}{4P} + (2+1) * t_{wait})$. If $t_{wait} < \frac{Nx}{8P}$ then the total running time

of the application is given by $\frac{Nx}{P} + t_{wait}$ and the application cost is given by $P * (\frac{Nx}{P} + t_{wait})$. This cost value is less than the cost value incurred in scenario I. Therefore, the extra overhead suffered by the application is not directly proportional to the number of scheduling operations. That is the reason for which the % cost increase was different for different distributions for the same problem size. Moreover, the overhead of the runtime system can also add to the load imbalance, in which case, the scheduling decisions can get affected. That could be reason for which the runtime system implementation of a particular scheduling technique was performing slightly better than the messaging layer implementation of the same scheduling technique as seen in the previous section.

## 5.4 Summary

The purpose of the chapter was to evaluate and analyze the overhead introduced by the runtime system PREMA. The main features of the DLBL library are one-sided communication and the remote service requests provided by PREMA. Since the DLBL library was built on top of PREMA which in turn was built on top the underlying messaging layer (MPI), it was almost certain the design will incur some overhead in addition to the scheduling overhead. It is not entirely possible to predict the amount of cost variation the application would suffer due to the use of PREMA. This is because, the overhead of PREMA can affect the scheduling decisions. Hence, in the first place, a small test program was written in order to estimate the latency and startup time of the runtime system PREMA. From the experiments conducted it was observed that the latency and the startup time are larger than the latency and the startup time of the messaging layer. Remote service requests was also found to be a costly operation when the amount of work involved with the handler is very small relative to the overhead. With these observations, we decided to analyze the performance of the runtime system in the real application. For this purpose, another implementation of the scheduling techniques using the messaging layer was developed. Since, the PREMA overhead is incurred during every scheduling operation, it was decided to run the comparison experiments for those techniques where $N_{sch}$ can be kept constant. The experiments were conducted on all the three test cases for the largest problem sizes and for various types of load distributions. Based on the experiments conducted, we observed that the % cost increase, on an average, was below 10% for *N-body* and *AQR* application. In some

cases, the PREMA implementation was even slightly better than the MPI implementation of the corresponding scheduling technique. However, for the *heat solver* application, the observation was very different. With increasing number of processors, the % cost increase was very high. This was attributed to the high latency and the high startup cost of the runtime system relative to the computational cost.

# CHAPTER VI

# CONCLUSIONS AND FUTURE WORK

In this last chapter of the thesis, a summary and an overall evaluation of the research work described in earlier chapters is presented. The conclusions and future research directions are also outlined.

An evaluation of the entire research work is presented in Section 6.1. In Section 6.2, lessons learned from this research work are presented. In Section 6.3, a number of directions for possible future research investigations are suggested.

## 6.1 Accomplishments

The primary objective of this thesis is: "to develop an application programming interface (API) that uses an integrated framework (dynamic loop scheduling and runtime system) for load balancing scientific applications on a distributed memory architecture. The emphasis is placed on enhancing the performance of the application by reducing the load imbalance among the processors caused by a wide range of factors".

In chapter I, a number of goals to be fulfilled in order to meet these objectives are listed. Refer to Section 1.4 for the goals of this thesis. The following paragraphs describe how the goals of this thesis have been fulfilled.

In chapter 2, a detailed review of the related areas of research namely the dynamic loop scheduling and the runtime systems is presented. The following concepts have been described there:

- The need for developing dynamic loop scheduling techniques.
- The state-of-the-art in various dynamic loop scheduling techniques.
- The various runtime systems and their features.
- The integrated approach by the Sun ANSI/ISO C compiler for shared memory systems.

- The commonly available static and dynamic partitioning tools for load balancing for parallel applications.
- The need for an integrated approach for distributed memory architectures.

In chapter 3, the detailed design of the API for the integrated approach is presented. The salient features of the API are:

- The one-sided communication protocol.
- The usage of remote service requests.
- The suite of dynamic loop scheduling techniques.
- The master-slave implementation of the API.

In chapter 4, a detailed description of the parallel benchmarks used for experimentation is given along with the interpretation of results. The parallel benchmarks considered are: the $N$-body simulations, the automatic quadrature routine (AQR) and an unstructured grid heat solver. Important performance highlights outlined in this chapter include:

- A qualitative analysis of dynamic loop scheduling techniques.
- A cost improvement of up to 75% for Gaussian distribution in $N$-body simulations over static chunking.
- A cost improvement of up to 76% for back distribution in AQR over static chunking.
- A cost improvement of up to 55% in unstructured grid heat solver over static chunking.

In chapter 5, the overhead of the runtime system is investigated. Some of the significant points are listed below.

- It is not possible to make a general characterization and identification of the amount of the overhead introduced by the runtime system. Instead, an analytical explanation has been presented.
- The latency and the startup time measurement of the runtime system vis-a-vis the native message passing layer. They are found to be a couple of magnitudes higher than the corresponding counterparts of the message passing layer.

- The remote service requests was found to be a costly operation, especially for the case where the amount of work involved with the handler is small when compared to the cost of execution of the handler.
- The overhead of the runtime system can add to the load imbalance which in turn can affect the scheduling decisions.
- A quantitative analysis of dynamic loop scheduling techniques is presented.
- For certain problem characteristics, the overhead of the runtime system was found to be less than 10% of the underlying messaging layer in terms of cost.

In chapter 2 and chapter 3, the accomplishment of the first three goals is presented. Goals 4, 5, and 6 are addressed in chapter 4. In chapter 5, the fulfillment of goal 7 is presented. Finally, goal 8 is discussed in the remaining sections of this chapter.

## 6.2 Lessons Learned

During the course of this research work, a number of interesting aspects have been observed:

- ***Performance of advanced techniques:*** The most advanced dynamic loop scheduling techniques like *adaptive factoring* were developed to address all the three sources of load imbalance. The performance of the advanced techniques underscores the results obtained from theory.
- ***Performance of Fixed size chunking:*** The *fixed size chunking* is an ideal tradeoff between *static chunking* and *self scheduling*. It exploits the good load balancing property of *self scheduling* and the low scheduling overhead of *static chunking*. However, in reality, it is not possible to find the optimal chunk size without any trial and error observation. This requires some knowledge about the application characteristics. If the optimal chunk size is found, then *fixed size chunking* can give the best performance. Therefore, estimating optimal chunk size is always application specific.
- ***Performance of Adaptive factoring:*** The *adaptive factoring* technique always depends on the initial chunk size in order to estimate the mean and standard deviation of the iterate

execution times. If the initial chunk size is greater than the optimal chunk size, then there may not be enough work left to smooth the unevenness caused by the earlier chunks. On the other hand, if the initial chunk size is lesser than the optimal chunk size, it is difficult to capture the exact mean and standard deviation of the iterate execution times which is indicative of the problem, algorithmic and systemic characteristics. Therefore, estimating initial chunk size is always application specific.

- ***Applicability of integrated approach:*** In spite of the advantages associated with the integrated approach, it is not always the panacea for all circumstances under the present conditions. There are applications, for instance, the unstructured grid heat solver which exploits the weakness of the runtime system, namely, the high latency and the startup cost. Unless these short comings are overcomed, the applicability of the integrated approach will always be limited.

### 6.3 Future Research Directions

The aim of this research work is to explore the possibility of the integrated approach, namely the dynamic loop scheduling techniques and the runtime system. Based on the experience and knowledge gained during this work, a number of questions have been raised. These questions lead to a number of general areas that show promise for possible future research work.

***Adaptivity:*** In an heterogeneous environment, the environment vary with time, for instance, the running conditions when execution of the problem started can be different from the current running conditions. This in turn can affect the performance of the scheduling technique if that particular technique is not able to tune to the changes in the environment. In such a situation, the ideal system should accommodate the changes in the environment. Solutions such as the ones using reinforcement learning techniques could be used to train the system to adapt to any change in the environment. To start with, the user could select a particular technique to be used for solving the problem. With the reinforcement learning techniques embedded into the system, the system will select the best scheduling technique based on the current environmental conditions.

***Estimation of chunk size:*** The optimal chunk size estimation for *fixed size chunking* and the initial chunk size estimation for *adaptive factoring* is application specific, and it can be estimated only through trial and error observation which may not be possible under all

circumstance because of the cost involved in running the application. Developing some kind of heuristic which helps in the estimation of chunk size is another area which we would like to explore.

## 6.4 Summary

In this chapter, the research work that was performed for this thesis work is summarized. This chapter consists of two main parts. In the first part, the research objectives and goals are discussed. The lessons learned from this research work along with the possible future direction is presented in the second part of the chapter.

## REFERENCES


[1] W. Gropp, E. lusk, and A. Skjellum, *Using MPI:Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA, 1994.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.

[3] S. Hummel, E. Schonberg, and L. Flynn, "Factoring: A method for scheduling parallel loops," in *Communications of the ACM, Vol. 35, No. 8*, pp. 90–101, 1992.

[4] S. Hummel, J. Schmidt, R. Uma, and J. Wein, "Load-sharing in heterogeneous systems via weighted factoring," in *Proceedings of Symposium on Parallel Algorithms and Architectures*, pp. 318–328, 1996.

[5] I. Banicescu and Z. Liu, "Adaptive factoring: A dynamic scheduling method tuned to the rate of weight changes," in *Proceedings of the High Performance Computing Symposium*, pp. 122–129, 2000.

[6] I. Banicescu and V. Velusamy, "Performance of scheduling scientific applications with adaptive weighted factoring," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium, Heterogeneous Computing Workshop*, 2001.

[7] P. Beckman and D. Gannon, "Tulip: A portable run-time system for object-parallel systems."

[8] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach, "CRL: high-performance all-software distributed shared memory," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, vol. 29, 1995.

[9] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel, "The mobile object layer: A run-time substrate for mobile adaptive computations," *Advances in Engineering Software*, vol. 31, pp. 621–637, 2000.

[10] N. Chrisochoides, I. Kodukula, and K. Pingali, "Data movement and control substrate for parallel scientific computing," in *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pp. 256–268, 1997.

[11] C. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1001–1016, 1985.

[12] C. Polychronopoulos and D. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1425–1439, 1987.

[13] I. Banicescu, V. Velusamy, and J. Devaprasad, "On the scalability of adaptive weighted factoring," *The Journal of Networks, Software Tools and Applications.*

[14] I. Banicescu and V. Velusamy, "Load balancing highly irregular computations with the adaptive factoring," in *Proceedings of the IEEE - International Parallel and Distributed Processing Symposium*, IEEE Computer Society Press, 2002.

[15] I. Banicescu and S. F. Hummel, "Balancing processor loads and exploiting data locality in $N$-body simulations," in *Proceeding of Supercomputing*, 1995.

[16] I. Foster, C. Kesselman, and S. Tuecke, "The nexus task-parallel runtime system," in *First International Workshop on Parallel Processing*, 1994.

[17] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *19th International Symposium on Computer Architecture*, (Gold Coast, Australia), pp. 256–266, 1992.

[18] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *IEEE Computer*, vol. 29, no. 2, pp. 18–28, 1996.

[19] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pp. 13–21, 1992.

[20] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained mobility in the Emerald system," *ACM Transactions on Computer Systems*, vol. 6, pp. 109–133, February 1988.

[21] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield, "The Amber system: Parallel programming on a network of multiprocessors," in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 147–158, 1989.

[22] B. Hendrickson and R. Leland, "The chaco user's guide: Version 2.0," Tech. Rep. Tech Report SAND94-2692, Sandia National Laboratories, July 1995.

[23] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," Tech. Rep. Tech Report CORR 95-035, University of Minnesota, 1995.

[24] K. Devine, B. Hendrickson, E. Boman, M. St.John, and C. Vaughan, "Design of dynamic load-balancing tools for parallel applications," Tech. Rep. Tech Report SAND99-1377, Sandia National Laboratories, 1999.

[25] J. L. Traff, H. Ritzdorf, and R. Hempel, "The implementation of mpi-2 one-sided communication for the nec sx-5."

[26] E. Luke, I. Banicescu, and J. Li, "The optimal effectiveness metric for parallel application analysis," *Information Processing Letters - Special Issue on Parallel Models*, vol. 66, no. 5, pp. 223–229, 1998.

[27] J.Barnes and P.Hut, "A hierarchical o(nlog(n)) force calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[28] A.W.Appel, "An efficient program for many-body simulations," *SIAM Journal of Computing*, vol. 6, pp. 32–39, 1985.

[29] L.Greengard and V.Rokhlin, "A fast algorithm for particle simulation," *Journal of Computational Physics*, vol. 73, pp. 325–348, May 1987.

[30] R.L.Carino, *Numerical Integration Over Finite Regions Using Extrapolation by Nonlinear Sequence Transformations.* PhD thesis, La Trobe University Australia, May 1992.

[31] R.L.Carino, I.Robinson, and E. de Doncker, "Adaptive cubature over a collection of triangles using the d-transformation," *Journal of Computing and Applied Mathematics*, vol. 50, pp. 171–180, 1994.