Profiling Architecture and Tools in Coven

A Thesis

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Computer Engineering

---

by

Vishal Patil

May 2005

Advisor: Dr. Walter B. Ligon III

May 6, 2005

To the Graduate School:

This thesis entitled "Profiling Architecture and Tools in Coven" and written by Vishal Patil is presented to the Graduate School of Clemson University. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Engineering.

Walter B. Ligon III, Advisor

We have reviewed this thesis
and recommend its acceptance:

Adam W. Hoover

Tarek M. Taha

Accepted for the Graduate School:

Abstract

Profiling can help make application parallelization more effective by identifying the performance bottlenecks. Visualization tools coupled with a tracing library can highlight the temporal aspect of performance variations, showing when and where the parallel program's performance is being compromised. A complex challenge is to analyze the data gathered during the program execution and provide useful suggestions to the user which shall enable him to make appropriate changes and thereby increase the throughput of the application as well as decrease the overall runtime. The following work details the profiling framework for Coven, a component based PSE toolkit. The framework consists of a modified tracing library used to log events, a GUI tool used to visualize the temporal aspects of the parallel program and a rule based expert system which analyzes the data gathered during the parallel program run and provides useful suggestions to the end user.

Dedication

To Mom and Dad

Acknowledgments

I would like to thank my advisor Dr. Walt Ligon III and Nathan Debardeleben for their guidance and support. I would like to thank the entire PARL group especially Will, Phil, Mike, Louis and Patrick for their invaluable help and support.

Table of Contents

Page

TITLE PAGE i

ABSTRACT ii

DEDICATION ii

LIST OF FIGURES vi

1 Introduction 1

1.1 Brief Overview of Coven 2
1.2 Motivation for Research 2
1.3 Proposed Solution 4
1.4 Thesis Overview 7

2 Related Work 8

2.1 Common Profiler Tools 9
2.1.1 Vampir 10
2.1.2 TAU 10
2.1.3 MultiProcessing Environment (MPE) 11
2.1.4 Paradyn 12
2.1.5 PAPI 13
2.2 Expert System Tools 13
2.2.1 PPA 14
2.2.2 Kappa-PI 15
2.2.3 PDE 15

3 Coven Profiling Framework 17

3.0.4 Coven model of Computation 18
3.1 Back end data collection service 19
3.1.1 Using MPE to log information 20
3.1.2 Determining the load and memory 26
3.2 Front end visualizer 27
3.2.1 Software Architecture 28
3.2.2 Visualizations 30
3.3 Rule Based Expert Agent 40

4 Case Studies 47

4.1 Work Environment 47

4.2 Comparing module run times . . . 48
4.3 Detecting memory problems . . . 50
4.4 Automatic Performance Analysis . . . 52
4.4.1 Enabling Multithreading . . . 52
4.4.2 Enabling Load Balancing . . . 54
4.4.3 Enabling Shared Memory . . . 56
5 Conclusions and Future Work . . . 62

BIBLIOGRAPHY . . . 65

List of Figures

Figure | Page

1.1 Coven Profiling Framework . . . 5

3.1 Sample Coven Application . . . 18
3.2 Sample code to profile function using MPE . . . 21
3.3 Mapping TPH flow to user defined state . . . 22
3.4 Log file visualization using Jumpshot . . . 27
3.5 Coven Profiler Software Architecture . . . 29
3.6 Pseudo Coven code for 2D FFT application . . . 31
3.7 Pseudo Coven code for N-Body application . . . 33
3.8 Total Module Runtime . . . 33
3.9 Total Event Runtime . . . 34
3.10 Module Thread Runtime . . . 35
3.11 Event Thread Runtime . . . 36
3.12 Load Information . . . 37
3.13 Memory Information . . . 38
3.14 TPH Lifetime View . . . 39
3.15 A sample JESS rule . . . 41
3.16 Before Multithreading . . . 42
3.17 After multithreading . . . 42
3.18 Without load balancing . . . 43
3.19 With load balancing . . . 43
3.20 Without shared memory, Coven uses MPI for inter-thread messages . . . 44
3.21 With shared memory . . . 44
3.22 Overlapping message transfers with module execution, without shared memory . . . 45
3.23 Overlapping message transfers with module execution, with shared memory . . . 45

4.1 Cannon's matrix multiplication algorithm . . . 48
4.2 Module Runtime before optimization . . . 49
4.3 TPH view before optimization . . . 49
4.4 Module Runtime after optimization . . . 49
4.5 TPH view after optimization . . . 49
4.6 N-body application with memory leak . . . 51
4.7 N-body application without memory leak . . . 51
4.8 Before Multithreading . . . 53
4.9 After Multithreading . . . 53

4.10 Before Load Balancing 54
4.11 After Load Balancing 56
4.12 Before Shared Memory 57
4.13 After Shared Memory 57
4.14 TPH view for FFTW application without shared memory 59
4.15 Load view for FFTW application with shared memory 59
4.16 TPH view for FFTW application with shared memory 60

5.1 Current load view for a module 63
5.2 Enhanced view for a module with computation, network I/O and disk I/O information 63

Chapter 1

Introduction

Problem Solving Environments (PSEs) have become an integral part of modern high performance computing due to the increase in the complexities of the types of simulations being run and the underlying problems being modeled, as well as the increasing complexity of the computer systems being employed. They often are targeted at a particular application domain, and have pre built tools and ready to use pieces of application code. These properties help encourage code reuse and quicker development time, especially during the design and test cycle. However, by abstracting away some of the complexities involved in developing scientific application codes, performance problems can be hidden and hard to detect, especially in parallel applications. Thus, a problem solving environment needs to provide performance analysis and profiling facilities in order to assist users in tuning their parallel applications for improved performance[17]. This thesis discusses how performance measurement and analysis has been integrated in Coven, a generic PSE toolkit and how these features assist a Coven programmer to make important decisions regarding the parallel program design in order to improve the performance and thereby decrease the overall runtime.

1.1 Brief Overview of Coven

Coven is a component-based framework used to develop parallel applications. The main goal of Coven is to shield a problem solver from the complexities of parallel programming thereby enabling him to concentrate on the problem that he is trying to solve. A parallel program in Coven consists of a collection of modules that are linked together to form a data flow graph[10]. A Coven module can be thought of as a software component that implements an algorithm. Each of the Coven modules implements a pre-defined interface which makes the modules reusable across different parallel programs. In addition to this the framework also provides the functionality to seamlessly multi-thread, load balance and checkpoint a parallel program without requiring any additional programming effort on behave of the problem solver[10].

1.2 Motivation for Research

Obtaining high degree of performance in a parallel application is a difficult task. Decisions for improving the performance have to be made by considering different kinds of information like the behavior of the programming model used, in order to select the most adequate primitives for the program. Moreover understanding the actual details of the parallel machine can also prove beneficial to comprehend the effect of using certain primitives in the processor and in the communication links. These requirements, although taken into account during the programming stages of the application, usually require an additional stage of performance analysis when the results obtained are far from the desired values. These performance values obtained from the analysis provide a measure of the quality of both the design and implementation of the parallel program [7].

When a parallel program developed using Coven does not perform as expected it is necessary to review it's behavior during the program runtime. Coven basically consists

of a middleware layer over which the parallel programs run. The fundamental problem with software systems using middleware packages is that the application writer no longer reasons at the system level best-suitable for performance debugging but rather at a higher level of abstraction. The middleware takes care of mapping the high level commands and directives to low level system calls, but when it comes to debugging or performance tuning, most middleware packages are not properly instrumented and therefore do not map the system state or the performance indicators back onto the higher level of abstraction related to their user API[25]. Performance measurement of the parallel programs written in Coven requires the development of instrumentation and analysis techniques beyond those used for the traditional MPI programs. Events pertaining to the Coven architecture need to be defined and these need to be tracked in context of the Coven language abstractions during the program execution.

Once the information pertaining to the monitored application has been gathered it is useful to depict this information in a manner that would enable the programmer to easily comprehend the behavior. Graphical visualization is a standard technique for facilitating human comprehension of complex phenomena and large volumes of data. Also the gathered data needs to presented in the context of the programming model being used. This helps the programmer in establishing a correlation between the performance information and the logical structure of the parallel program. For example, it might be useful to illustrate the execution time of the different Coven modules that make up the parallel program. This can help the programmer in comparing the performance of different modules thereby enabling him to detect a module thats needs to be tuned so as to optimize the overall parallel application.

The parallel programmers using the visualization tools to study the behavior of the parallel program need substantial expertise and effort. The large amount of performance information offered by these tools can sometimes inundate a parallel programmer thereby making it difficult for him to make decisions for improving the

parallel program design. In addition, most of the parallel programmers are not skilled in identifying and solving performance problems. However the chief goal of Coven is to make things as simple as possible for a parallel programmer thereby enabling him to concentrate on the problem that he is trying to solve. Most of the parallel programs that need optimization fall into one of two categories. The first consists of those for which good optimization strategies can be derived from well known heuristics. For such programs an automated expert system for guiding optimization can potentially achieve good results. The second category consists of programs which need new algorithms and techniques for optimizing them. For such programs the intervention of a human programmer is obviously needed. However there are many parallel applications that have well-recognized performance problems for which optimization can be found using a few good heuristics[14]. Automating program optimizations using knowledge based systems can significantly help reduce the parallel software development effort. Even in cases where complete automatic techniques are not possible, it is beneficial to automate the optimization steps to extent possible.

1.3 Proposed Solution

In this thesis we propose a profiling framework that facilitates the task of a programmer to understand the behavior of parallel programs written in Coven and if possible provide suggestions to improve their runtime. As shown in the figure 1.1 the framework consists of the following three important components:

- A data collection service which gathers information regarding the important events during the life time of the parallel program.
- A visualization tool that uses the gathered information to highlight the important performance aspects of the program.

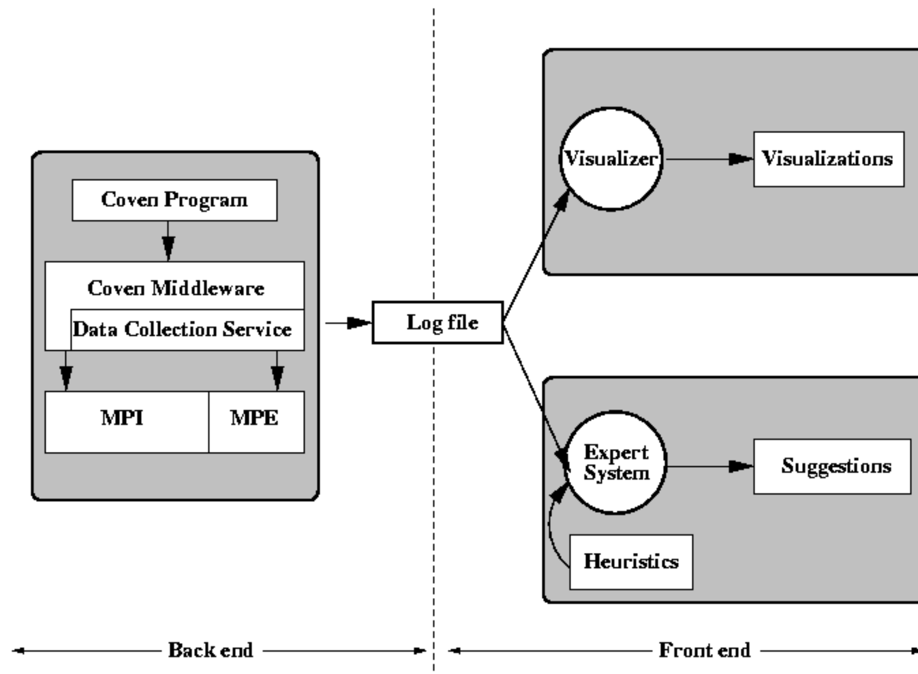

Figure 1.1: Coven Profiling Framework

- An expert system which uses heuristics to process the gathered data and provides suggestions to improve the performance of the program.

The **data collection service** which is a part of the Coven middleware, is a tracing library that gathers information regarding the different events that occur during the lifetime of the parallel program. In order to log this information the Coven programmer does not have to manually instrument the parallel program, instead the Coven middleware performs the data flow analysis of the program and identifies the appropriate instrumentation points. The information regarding these points of interest is then automatically logged by the data collection service during the parallel program run. The instrumentation and tracing employed by the data collection service is based on the tracing library provided by MPE (MultiProcessing Environment)[5], which is used to log information regarding MPI calls in a parallel program. MPE also enables a programmer to define his own events and log them. This feature has been used to define events pertaining to the Coven framework in terms of the programming

language constructs and these get logged during the parallel program run. At the end of the parallel program execution, the MPE library handles the functionality of gathering the trace information from all the nodes of the cluster and coalescing this information into a single binary log file.

The binary trace file generated by the monitored parallel program contains time sequenced information regarding the different events that occurred during it's life time. These events can then be studied using Jumpshot [26], a visualization tool provided along with MPE. However this tool renders information in terms of MPI calls and not in terms of the Coven programming constructs. Hence in order to study the history of the program run in terms of the Coven programming model, a **visualization tool** (Coven profiler) has been developed which depicts different aspects of the parallel program run. The tool can be used to study different performance metrics such as the execution time, memory usage and load exerted in terms of the chief Coven programming constructs namely modules and threads. The GUI tool has been developed using the Java Swing API[24] and a modified *slog2sdk*[4] library, which provides a Java API to extract information from the binary log file generated by MPE. In addition to visually displaying the performance information, the tool enables the user to easily obtain detailed information regarding the different performance parameters during the lifetime of the parallel program. This is achieved by permitting the programmer to query performance information at a particular instance of time during the program runtime by selecting a pertinent point on the graph. In case of such event the tool extracts the relevant information from the log file and displays it in a manner useful to the programmer.

To assist the programmer in analyzing the performance problem a **rule based expert system** has been developed using JESS (Java Expert System Shell) [9]. The expert system uses some predefined heuristics and the information extracted from the trace file to perform automatic performance analysis of the parallel program run.

The expert agent then provides suggestions to the user with the aim of decreasing the parallel program runtime. The rules used to make decisions regarding the behavior of the parallel program are encoded in the form of text based scripts, hence changing and testing the heuristics simply entails editing of the script file and does not require any recompilation of the source code.

The profiling architecture has been tested by studying the behavior of several parallel applications implemented using Coven. These applications comprised of well known parallel algorithms (like 2D-FFT and Cannon's parallel matrix multiplication) as well as synthetic applications. The data generated by these applications was visualized to understand their behavior and if possible identify performance bottlenecks. In addition to interpreting the program behavior using the Coven profiler, the expert system was consulted to obtain suggestions which would improve the runtime of the applications. The changes suggested by the expert system were then made and the parallel applications were rerun to verify the improvement in performance.

1.4 Thesis Overview

The work presented in this thesis dissertation has been divided into five chapters. The second chapter defines the problem of performance analysis of parallel programs and the common methods of confronting this problem using existing monitoring tools and knowledge based systems. The third chapter gives a detailed explanation of the software framework pertaining to the profiling architecture in Coven which consists a modified tracing library, a Java based visualization tool and a rule based expert system which detects common parallel program problems. The fourth chapter explains the results of analysis of some parallel applications. These applications have been chosen to represent the important categories of parallel message passing applications. Finally in the fifth chapter the conclusions and future lines of the work are presented.

Chapter 2

Related Work

The traditional reason for using parallel or cluster computing is one of computational limitations, e.g. the complexity of the calculations is limited by the computational time. Gaining the best performance from the system is thus essential. An overall assessment of the code can be gained by simply inserting timing calls within the code. However much more detailed information is often required and inserting hundreds of timing calls within a code can be extremely tedious. An alternative is to use a profiler, which should provide more detailed timing information without code modification. There are a number of serial profilers available, two of the most common being prof and gprof. These tools provide timing information for various routines and are useful for isolating computationally intense parts of a serial code. Parallel codes however contain interprocessor communications and the user is interested in these, as well as the computation[22]. Hence specific parallel profiling tools have been developed by numerous groups and we discuss a few of these in the first half of the chapter.

Current state-of-the-art profiler tools provide valuable assistance in analyzing the performance of parallel programs by visualizing the runtime behavior and calculating statistics over the performance data. However, the developer of parallel programs is still required to filter out relevant parts from a huge amount of information shown in

numerous displays and map that information onto program abstractions without tool support. Attempts have been to made to automate the process of the performance problem detection either by conducting a control flow analysis of the parallel program or by post mortem analysis of the data generated by the parallel application. The second half of the chapter looks at some of the attempts that have been made to solve the problem of automatic parallel performance problem detection.

2.1 Common Profiler Tools

Most of profiler tools instrument the parallel application inorder to generate a trace file containing the program runtime information. Instrumentation normally needs the addition of some extra calls to subsystem that gives a timestamp and records the execution of every instrumented primitive. On running the monitored application, each instrumented primitive that executes generates an event which gets logged to the trace file. Information included in the events depend on the actual format of the trace but it usually contains a timestamp, an identification of the executing process and some details about the instrumented primitive[7].

The trace file information is generally used to show the evolution of the execution to the user in graphical analysis session. Typical graphical views include GANTT chart, where the activity of each processor is represented by a horizontal line. Different activities are depicted by different colors. Other views are also used to express the dynamic behavior of the programs: bar graphs or pie charts are just a few examples of rich variety of graphical data views.

In the following section, we shall discuss some of common parallel profiler tools that are used to study the behavior of parallel programs.

2.1.1 Vampir

Vampir coupled with VampirTrace is a performance analysis tool for MPI programs written in C, C++ and Fortran. The Vampir visualization tool enables the user to analyze various aspects of the runtime behavior of message-passing programs. It displays post-mortem trace files in a variety of graphical views, and provides flexible filter and statistical operations that condense the displayed information to a manageable amount. It excels in rapid zooming, allowing the user to quickly focus on arbitrary time intervals. Thus performance bottlenecks can easily be identified and investigated at the appropriate level of detail. Vampir can display multiple views of the parallel application execution, each one presenting information in a distinct way. This includes a GANTT chart which shows per process application activities and message passing along time axis. Source code click back is avaliable on platforms with compiler support. Other displays include statistical analysis of program execution, statistical analysis of communication operations, and a dynamic calling tree display. VampirTrace is a MPI profiling library that generates Vampir traces. It hooks into the MPI profiling interface, and guarantees low instrumentation overhead. The effects of distributed clock drift is automatically corrected. Tracing can be controlled dynamically during runtime to minimize the amount of trace data to be collected[18]. However Vampir and VampirTrace are commercial products and can be used freely only for a limited amount of time.

2.1.2 TAU

The TAU performance framework is composed of instrumentation, measurement, and visualization phases. TAU supports a flexible instrumentation model that allows the user to insert performance instrumentation calling the TAU measurement API at several levels of program compilation and execution stages. The instrumentation identifies code segments, provides mapping abstractions, and supports multi-threaded and

message passing parallel execution models. Instrumentation can be inserted manually, or automatically with a source-to-source translation tool. When the instrumented application is compiled and executed, profiles or event traces are produced. TAU can use wrapper libraries to perform instrumentation when source code is unavailable for instrumentation. Instrumentation can also be inserted at runtime, using the dynamic instrumentation system DynInst[6], or at the virtual machine level, using language supplied interfaces such as the Java Virtual Machine Profiler interface. The instrumentation model of TAU interfaces with the measurement model which can be sub-divided into two models. A high level model which determines how events are processed and a low-level measurement model that determines what system attributes are measured. By providing a flexible measurement infrastructure, a user can experiment with different attributes of the system and iteratively refine the performance of a parallel application. TAU comes with both text-based and graphical tools to visualize the performance data collected. The graphical tool used to visualize the trace is called RACY while the text based profiler is termed as pprof. In addition to this it provides bridges to other third-party tools such as Vampir [18] for more sophisticated analysis and visualization. The performance data format is documented and TAU provides tools that illustrate how this data can be converted to other formats [8].

2.1.3 MultiProcessing Environment (MPE)

MPE is a software package distributed along with MPICH[16]. It consists of a tracing library and tools that are use to understand the behavior of parallel programs implemented using the MPI library. Also MPE may be used with any implementation of MPI[5]. When a MPI program is linked with the tracing library provided with MPE all the standard MPI calls get logged and are stored in a binary file at the end of the program execution. The record pertaining to each MPI call which gets logged consists of the MPI function name, a time stamp and the id of the parallel

task making the call. In addition to logging the standard MPI calls, it is possible for a programmer to define his own events and log information pertaining to those events. The functionality of gathering the log information from the different nodes of the cluster and coalescing it into a single binary log file is handled completely by the MPE library. The log file generated can be either in ALOG, CLOG or SLOG format, with SLOG being the most scalable format. In order to understand the behavior of the parallel program using the generated binary log file, MPE provides a visualization tool call Jumpshot[26]. This tool displays the various events that occur during the parallel program run in the form of a GANTT chart.

2.1.4 Paradyn

Paradyn uses several novel technologies so that it scales to long running parallel programs (hours or days) and large (thousand node) systems, and automates much of the search for performance bottlenecks. Paradyn is based on a dynamic notion of performance instrumentation and measurement. Unmodified executable files are placed into execution and then performance instrumentation is inserted into the application program and modified during execution. Paradyn can gather and present performance data in terms of high-level parallel languages (such as data parallel Fortran) and can measure programs on massively parallel computers, workstation clusters, and heterogeneous combinations of these systems. Paradyn provides a simple library and remote procedure call interface to access performance data in real-time. Using this library it is possible to interface existing performance visualization systems like PABLO[20]. Visualization modules in Paradyn are external processes that use the visualization library and interface. All performance visualizations are implemented as visualization modules. Paradyn currently provides views in the form of time-histograms (strip plots), bar charts, and tables. In addition to visualizing the parallel program execution Paradyn helps in performance problem detection using a search

model called W^3. The W^3 search model tries to answer three questions: why is the application performance poor (identifying the type of bottleneck, e.g., synchronization, I/O, and computation)? where is the bottleneck (isolating a performance bottleneck to a specific resource, e.g., a synchronization variable, a disk system, or a procedure)? and when does the problem occur (isolating a bottleneck to a specific phase of the program's execution) [13]. It also uses the same model to guide placement and modification of dynamic instrumentation[6].

2.1.5 PAPI

PAPI (Performance Application Programming Interface) is an effort to establish a uniform, standard programming interface for accessing hardware performance counters on modern microprocessors[2]. Hardware performance counters can be very useful for tuning the performance of applications and for evaluating the effectiveness of the compiler on the application. These counters allow direct measure of the actual usage of the hardware as an application runs and may help to diagnose bottlenecks in the application's performance. PAPI provides two standardized APIs to access the underlying performance counter hardware, a low level interface designed for tool developers and expert users and a high level interface is for application engineers. Using cross-platform interface to the counters, allows maintenance of a common source for a wide variety of architectures. TAU as well as Paradyn provide an interface to use PAPI to gather low level information regarding the parallel applications[8, 6].

2.2 Expert System Tools

Most of the tools listed above rely on fair degree of graphical representation of the parallel program to help the programmer in detecting the performance bottlenecks. However for large parallel applications the visualization of the program flow can be

too confusing to help the user to draw any conclusions. Moreover, performance prediction and performance analysis based on accurate evaluation techniques like analytical model techniques is not supported in the above tools. Attempts have been made to overcome this problem by using Artificial Intelligence (AI) based techniques to automatically detect the performance problems. These techniques treat the problem of finding performance bottleneck as a search problem[15]. The expert system tools define a set of rules that test for possible causes of the performance loss and on recognizing these make appropriate suggestions to the user.

The following sections of the chapter discusses examples of knowledge based tools used to identify performance problems pertaining to parallel applications.

2.2.1 PPA

PPA (Parallel Program Analyzer) uses the program analysis techniques to detect the strategy and algorithm concepts in a parallel program and suggests a better strategy, if there is one, to the programmer. Since PPA derives algorithm concepts from the text of a program by statically examining its source code without using any specification or execution information, the problem of overload of trace data for large scale parallel programs does not exist. PPA uses object oriented techniques to represent programming concepts, plan based techniques to represent the program knowledge, and a deductive inference framework to derive the algorithm concepts from the program structure. The PPA tool consists of three important components namely an event base, a plan base and an inference engine. A program parser scans the source code and represents the statements in terms of events and stores them in the event base. The plan base is a knowledge base in which plans are represented as inference rules. The inference engine is the reasoning component which repeatedly applies the rules in the plan base to the events in the event base until no more events

can be derived from the existing ones and presents the user with observation and and recommendations of algorithmic optimization[15].

2.2.2 Kappa-PI

The main objective of the Kappa-PI (Knowledge-based Analyser of Parallel Program Applications and Performance Improver) tool is to give parallel programmers some aid when analyzing the performance of their applications[7]. This tool makes an automatic post mortem analysis of the program behavior and provides suggestions regarding the behavior of the program. The Kappa-PI tool is currently implemented to analyze the parallel applications developed using PVM[21]. The tool bases the search of the performance problems on it's knowledge of their causes. The tool makes a *pattern matching* between those execution intervals with degraded performance and it's knowledge base of the cause of the problems. This is the process of identification of the problems and creation of the recommendations for their solution. This working model allows the performance problem database to adapt to new possibilities of analysis with incorporation of new problems (new knowledge data) and new types of programming model[7].

2.2.3 PDE

PDE (Program Development Environment) is a tool for programming distributed memory system. The main goal of this tool is to handle all of the complexity introduced by the parallel hardware such as data or load distribution or data communication in a user-transparent way. This has been achieved by offering the programmer a high level, domain specific specification formalism, an expert system to exploit parallelism and collection of hierarchically organized algorithmic skeletons. An algorithm skeleton consists of a basic, configurable structure for efficient parallel execution of a small related algorithm class on a particular parallel hardware. The initial problem

specification is divided into two parts, purely computational features and features relevant to parallel structures. The first is passed to the program synthesizer while the latter goes to the programming assistant. The programming assistant is an expert system developed using CLIPS[1] which performs the task of analyzing the problem specification given by the user, extracting the relevant parts for the parallel framework and the selections of suitable algorithmic skeletons. The program synthesizer expands the parallel framework into compilable, hardware specific, parallel C++ and C programs[11].

Chapter 3

Coven Profiling Framework

The chief goal of this research is to develop a performance analysis framework that will help expert as well as novice parallel programmers to understand the behavior of parallel programs developed using Coven. These tools need to enable the programmer to visualize the important aspects of the parallel program as well as provide a list of suggestions that could possibly improve the performance of the application.

The proposed framework can be divided into the following three important components namely

- Back end data collection service
- Front end visualizer and
- A rule based expert agent

However before explaining the proposed framework in detail, it would be beneficial to have a brief idea about the Coven computational model. This would help in highlighting the reasons why we chose a particular approach to implement the performance analysis framework.

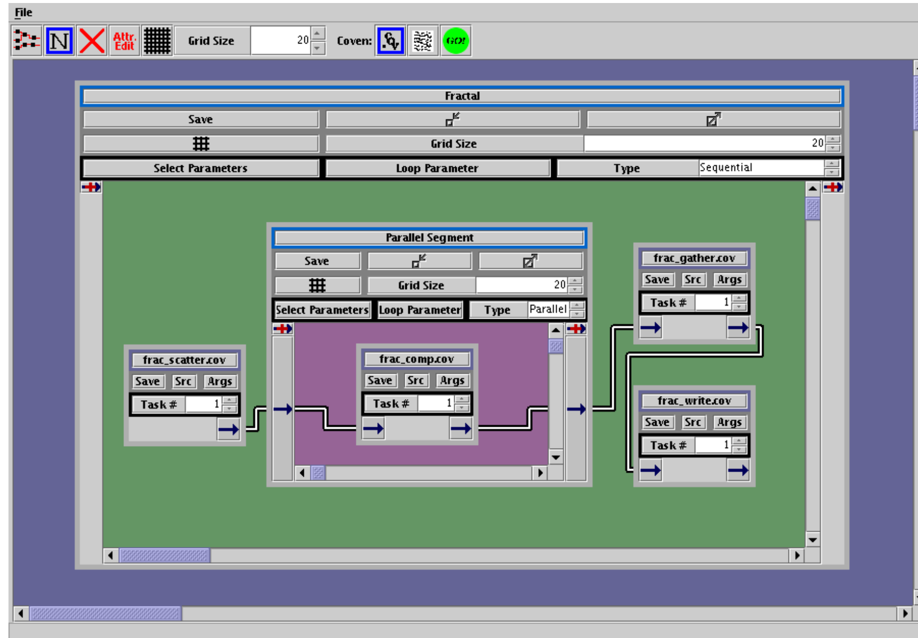

Figure 3.1: Sample Coven Application

3.0.4 Coven model of Computation

As explained earlier a parallel program in Coven consists of a collection of modules linked together to form a data flow graph. Figure 3.1 shows an example of a Coven program design for a parallel fractal application. The rectangular blocks represent the Coven modules while the pipes connecting them indicate the flow of data from one module to another. Each of the Coven modules essentially runs in a Coven program thread. The Coven master thread runs on the head node of the cluster while the slave threads run in parallel on the slave nodes of the cluster, with one or more slave threads running on the same node. Using the graphical program editor the parallel programmer can specify which modules need to be executed serially in the master thread and which need to be executed in parallel in the slave threads by enclosing these modules within a "parallel segment" block as shown in the figure.

Coven's runtime engine (middleware) which has been implemented using the MPI library is responsible for running the user application. It is responsible for operations such as: data encapsulation, module execution, memory maintenance and the

movement of data between the modules. All of these functionalities have been implemented in the runtime engine with the help of an internal data structure called Tagged Partition Handle (TPH). This data structure is completely transparent to the user. All the data in Coven that moves from one module to another is contained inside a TPH. Whenever a module reads data as input, it reads it from the TPH and when ever it creates a new output, it creates it inside a TPH. TPHs can be considered analogous to packets in a network. Much like packets, TPHs contain arbitrary data and are wrapped up into a data structure that flows around the system. The header information of the TPH contains details like where is the TPH destined as well as hints to what kind of information is contained within it. Similarly, much as a packet flows around a network between hosts, TPHs pass between modules and parallel processes. The Coven component responsible for calling the modules and passing the TPHs between them is called as the program sequencer. Each parallel task executes a single program sequencer[10].

The following sections shall now explain each of the performance framework components in detail with reference to the Coven model of computation.

3.1 Back end data collection service

When a Coven parallel program is executed, information regarding the important events pertaining to the Coven runtime as well as the parallel program get logged by the data collection service. At the end of the parallel program run all of this information is gathered at the head of the cluster and is stored in a binary log file. In the following section we shall discuss how the data collection service is implemented using MultiProcessing Environment (MPE). This is followed by a brief explanation about how the load and memory information is determined for a parallel task at a node of the Linux Beowulf Cluster.

3.1.1 Using MPE to log information

Coven makes use of the tracing library provided by MPE (Multi-Processing Environment) [5] for profiling the parallel applications that run under it's framework. MPE provides a number of useful facilities to the MPI programmers. These include several profiling libraries used to collect information about MPI programs and generating log files for post-mortem visualization. Also MPE may be used with any implementation of MPI[5].

The MPE tracing library provides a function **MPE_Log_event** which is used to log timestamped events. The MPE_Log_event function stores the information pertaining to the timestamped events[3] in memory, and these memory buffers are collected and merged into a single binary file at the end of the parallel program execution. The calls to MPE_Log_event are made automatically for each MPI call when a parallel program is linked with the MPE library[5].

In addition to logging timestamped events, MPE logging calls can be inserted into user defined functions to define and log states. These states are termed as **user defined states**. Logically a state can be thought of as a data structure that is logged to the binary log file and which contains the following information

- Time stamp corresponding to the start of the function
- Time stamp corresponding to the end of the function
- The function name
- A color associated with the state (used for visualization)

The routines **MPE_Describe_state** and MPE_Log_event are used to describe and log user-defined states. The use of these routines to profile a sample C function in a MPI program is illustrated in figure 3.2. In addition to logging the timestamps, the

```
int eventID_begin, eventID_end;
...
eventID_begin = MPE_Log_get_event_number();
eventID_end   = MPE_Log_get_event_number();
...
/* Specify the User defined state */
MPE_Describe_state( eventID_begin, eventID_end,
                           "Amult", "bluegreen" );
...
Amult( Matrix m, Vector v )
{
    /* Start of the user defined state*/
    MPE_Log_event( eventID_begin, m->n, (char *)0 );

    ... The entire function code ...

    /* End of the user defined state*/
    MPE_Log_event( eventID_end, 0, (char *)0 );
}
```

Figure 3.2: Sample code to profile function using MPE

MPE_Log_Event enables a programmer to log a 16 byte string and a 32 bit integer information along with each timestamped record.

The Coven runtime engine uses the concept of *user defined state* to log information regarding the various events that take place in the Coven runtime. Figure 3.3 shows how the concept of user defined state is used track the flow of a TPH through a Coven module. The start of the user defined state is used to log information regarding the entry of the TPH into a module while the end of the user defined state is used to log information regarding the exit of the TPH from a module. With each timestamped event record corresponding to the entry and exit of the TPH from a module, the processor id, Coven thread id, TPH id and other additional information like load and memory consumption are saved in the 16-byte string. Thus each user defined state stored in the binary log file contains the following information

- TPH id

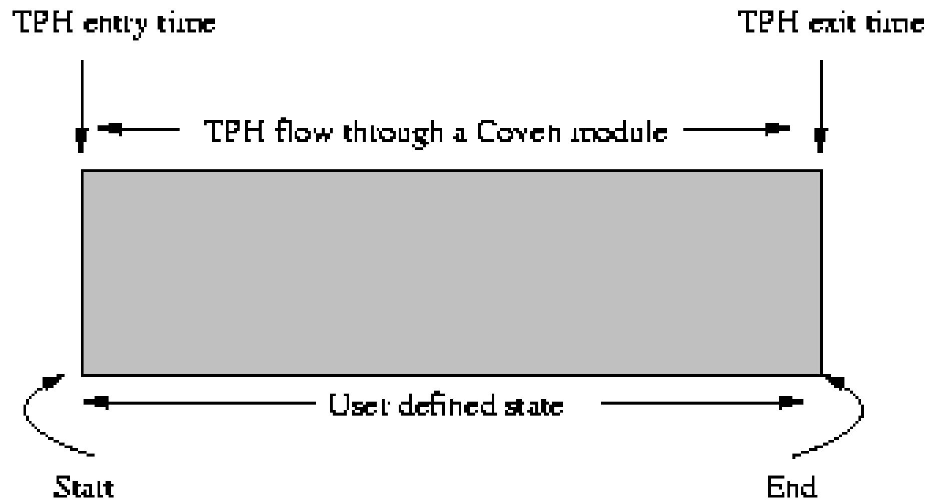

Figure 3.3: Mapping TPH flow to user defined state

- Module id
- The thread id
- Time when the TPH enters the module
- Time when the TPH exits the module
- Load exerted by the module

Storing information regarding the flow of TPHs in the parallel program is the most important function of the data collection service. This stored information enables offline review of the steps that occurred during the Coven program run using the Coven profiler. In addition to storing data regarding the flow of the TPHs, user defined states are used to log information regarding the different events related to the Coven runtime. These events pertaining to the Coven middleware are known as the Coven auxiliary events.

Coven auxiliary events

The Coven auxiliary events are logged in order to understand the behavior of the Coven middleware. Normally the information regarding these events is filtered out

by a Coven programmer using the Coven profiler, since he is more interested in understanding the behavior the parallel application than the behavior of the framework. However the information regarding these events can prove useful to the system programmers who are involved in the development of the Coven framework. The following events have been defined to understand the behavior the middleware:

- **Startup** : This is a user defined state used to determine the amount of time required for the initialization of the Coven middleware.

- **Program Sequencer** : The Coven component responsible for calling the modules and passing the TPHs between them is called as the program sequencer. Each parallel task executes a single program sequencer. A user defined state has been defined to log the start and end times of the program sequencer running on each of the slave nodes of the cluster.

- **Cleanup** : This user defined state logs the timing information regarding the steps that the Coven middleware executes, once it has finished executing a parallel program.

- **Sleep / Wait** : When there are no TPHs available to be processed by the program sequencer, it is put to sleep. This user defined state is used to determine the amount of sleeping time for the program sequencer.

- **Process TPH** : Whenever a TPH arrives for processing, the program sequencer executes the modules assigned to it in a sequential manner, passing the TPH into each module as an argument. The "Process TPH" user defined state is used to log the start and end times of the TPH processing by the program sequencer.

- **Virtualized Send** : Coven supports the concept of virtualization in which a Coven thread sends or receives a message without specifying what thread it's communicating with, instead it simply specifies what part of the data to talk

to[10]. The "Virtualized Send" event is used to log the time when a Coven thread sends a message using virtualization.

- **Virtualized Receive** : This auxiliary event is used to log the time when a Coven thread receives a message using virtualization.

- **TPH Created** : This auxiliary event is used to log the time when a TPH is created.

- **TPH Destroyed** : This auxiliary event is used to log the time when a TPH is destroyed.

- **TPH Network Send** : The Coven middleware uses asynchronous MPI send and receive calls to transfer TPHs. As TPHs arrive in the output queue of a programmer sequencer, an asynchronous MPI send is issued containing the TPH data and a control header. This event is used to log the time when the asynchronous MPI send has been issued.

- **TPH Network Receive** : This event is used to log the time when a TPH arrives at a program sequencer for processing.

- **TPH Enqueue** : Each program sequencer has an input queue which is used to store the TPHs that need to be processed by the modules assigned to the program sequencer. The "TPH Enqueue" auxiliary event is used to log the time when a TPH is enqueued in the input queue of the program sequencer.

- **TPH Dequeue** : Similarly, each program sequencer has an output queue which stores the TPHs once they have been processed by the modules assigned to the program sequencer. The "TPH Dequeue" auxiliary event is used to log the time when a TPH is dequeued from the output queue.

- **Memory Now** : This event is used to log information regarding the amount of memory being consumed by a Coven program thread.
- **Requesting Steal** : Internally Coven employs the Random Stealing dynamic load balancing algorithm in order load balance a parallel program. Whenever a Coven task runs out of TPHs to process, it issues a broadcast ”steal request”. The ”Requesting Steal” auxiliary event is used to log the time when a steal request is issued.
- **Sending Steal** : When a steal request is issued, heavily loaded tasks can choose to respond by sending a portion of their work to be processed by the stealer. The ”Sending Steal” event is used to log the time when heavily loaded task sends a portion of it’s work to the requesting parallel task.

Log file format

MPE provides the option of storing the profiling information either in ALOG, CLOG or SLOG binary format. Unlike the ALOG and CLOG binary log file format, the SLOG format is most scalable format since it does not require the entire log file to be parsed and stored in the memory. However currently SLOG strips off the 16 byte string and 32 bit integer from each time stamped event record which is used to store the following additional data (exported by Coven the framework)

- Processor id
- Coven thread id
- TPH id
- Module load
- Memory consumption

However this required extra information can be encoded along with each timestamped event record in the CLOG file format in the form of a 16 byte string. Since this information is essential to understand the functionality of the parallel program the CLOG file format has been chosen as the default log file format for the Coven framework.

3.1.2 Determining the load and memory

The MPE tracing library does not support the determination of load and memory consumption for a parallel task. In Coven, this information is extracted from the *Linux /proc file system*. If **pid** is the process id corresponding to the parallel task whose load and memory consumption is to be determined, then the statistical information pertaining to this task can be extracted from the */proc/***pid***/stat* file. This file contains the following information useful to Coven

- The amount of memory being consumed by the process.
- The amount of time for which the process has been scheduled on the CPU in the system (kernel) mode (*system time*)
- The amount of time for which the process has been scheduled on the CPU in the user mode (*user time*)

The *system time* and *user time* values are used to determine the *load* for the module as follows

$$\textit{scheduled time} = \textit{system time} + \textit{user time}$$

The percentage load exerted by the module can then be calculated as

$$\textit{load} = \textit{(scheduled time/total time)}$$

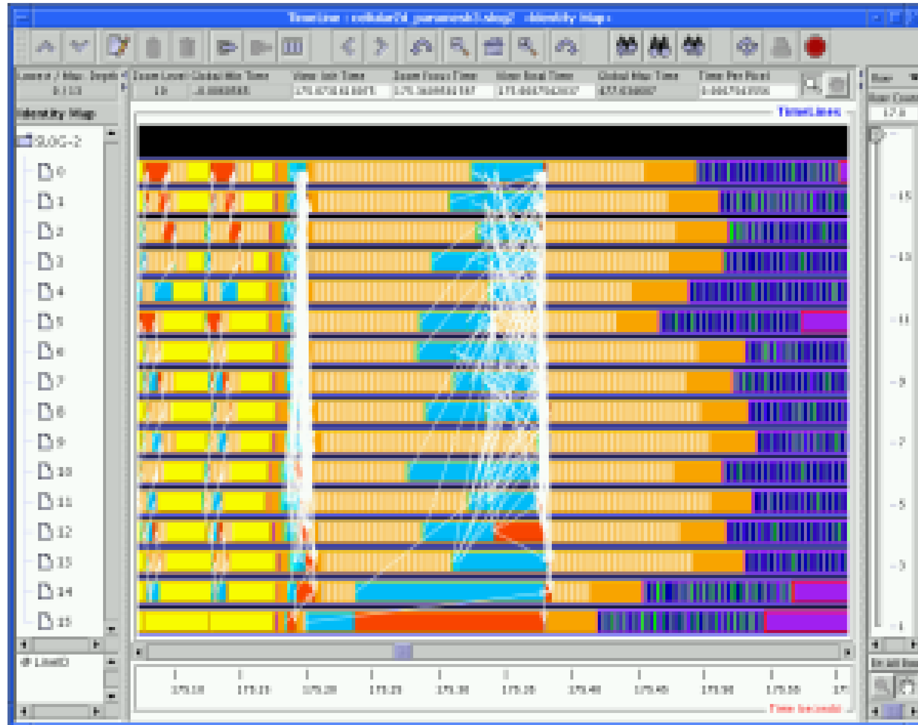

Figure 3.4: Log file visualization using Jumpshot

Where *total time* is the time in entirety (which includes CPU, network I/O and disk I/O time) for which the Coven module is scheduled by the Coven middle-ware for execution.

3.2 Front end visualizer

The log file generated by the Coven framework can be visualized using the Jumpshot visualizer that accompanies MPE. As seen in figure 3.4, jumpshot provides a single view which consists of a *timeline window* in which time (in seconds) is indicated on the x-axis and MPI process ranks are shown on the y-axis. Colored rectangles spanning sections of the time lines indicate that a particular process was in a particular state during the indicated time interval. These states are defined by the logging library and typically consist of MPI function call durations and the durations of user-defined states. Clicking with the mouse on such a rectangle pops up a small window containing

detailed information (state name, precise duration, etc). The arrows connecting the states represent the MPI messages. Details about a particular message (length, tag, etc.) appear when one clicks on the small circle appearing near the origin of the arrow[26].

There are several other aspects of a parallel program like load and memory that cannot be studied using Jumpshot. Moreover, Jumpshot was specifically developed to view the behavior of MPI programs and is unable to display the parallel program runtime information in terms of Coven programming constructs. These shortcomings lead to the development of a custom front end visualizer for the Coven log files. This tool is the **Coven Profiler** and was developed to provide the following information about a parallel program written in Coven:

1. The flow of TPHs during the program execution
2. Computational load exerted by the different Coven modules
3. Memory consumption of each Coven thread
4. The amount of time spent executing a Coven module in each program thread
5. The cumulative time spent executing a Coven module

The tool was developed using the Java Swing API [24]and a modified version of the **slog2sdk**[4] which is a part of MPE and provides a Java based API to parse the CLOG binary file. This API had to be modified to extract the Coven specific information (mentioned in section 3.1.1) from the CLOG file.

3.2.1 Software Architecture

The Coven profiler has been implemented using the **Model View Controller** design pattern. The advantage of using this pattern is that it completely decouples the data from the visualizations. This can prove to be beneficial in case the format of the log

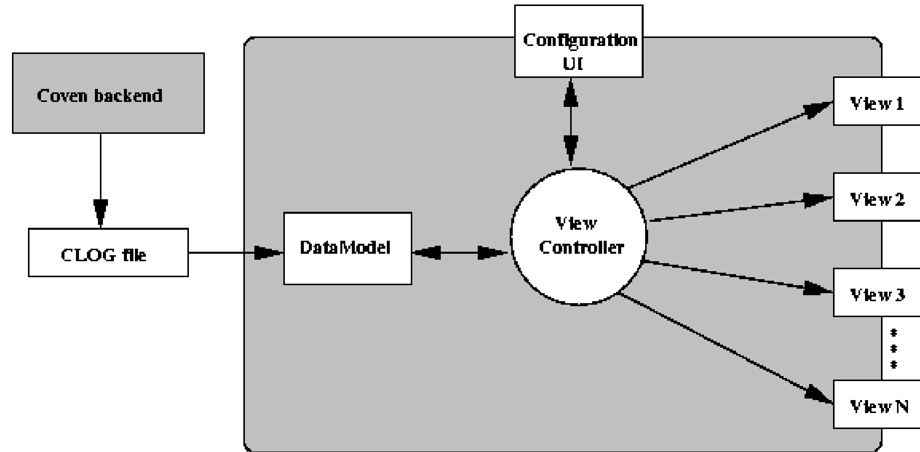

Figure 3.5: Coven Profiler Software Architecture

file used to store the information is changed. As seen in figure 3.5 the profiler consists of three important components: data model, controller and the views. Each of these components is explained in brief below.

Data Model

This component is responsible for parsing the input log files and initializing the various internal data structures which are used by the views to graphically render the data. The Data Model is constructed by implementing a simple interface defined by the profiler. The profiler then uses this interface to extract information from the log file through the model. The advantage of this approach is that it is possible to use a another log format without affecting the rest of the system. This can be done by simply defining a new data model for the log format and implementing the interface defined by the profiler. The current data model is implemented using the slog2sdk[4] which is used to parse the CLOG files.

View Controller

This component is responsible for controlling the flow of data from the Data Model to the various visualizations. After the information is extracted from the log files and

stored in the internal data structures, the View Controller uses them to initialize the various visualizations. Working in coordination with the Configuration UI, the View Controller is also responsible for reducing the information presented to the user as they refine the trace data they are interested in analyzing.

Views

These form the visualizations that depict the different aspects of a parallel program run. The profiler also provides an interface to incorporate new visualizations. Creating new views requires simply implementing this interface, upon which the view can interact with the Data Model API. The profiler tool currently offers the following seven visualizations to the programmer to understand the behavior of a parallel program written in Coven.

1. Total Module Runtime View
2. Total Event Runtime View
3. Module Thread Runtime View
4. Event Thread Runtime View
5. Load Information View
6. Memory Consumption View
7. TPH Lifetime View

3.2.2 Visualizations

Each of the seven visualizations mentioned earlier shall now be explained in detail in the following sections with the help of the 2D-FFT (Two Dimensional Fast Fourier Transform) and N-Body Coven applications.

```
. . .
   thread_master fftw_scatter(args)
      PARALLEL {
       for(i=0;i<3;i++){
          thread_slave fftw_comp_nd(args)
          thread_slave fftw_transpose(args)
          thread_slave fftw_comp(args)
          thread_slave fftw_transpose(args)
       }
     }
   thread_master fftw_gather(args)
. . .
```

Figure 3.6: Pseudo Coven code for 2D FFT application

2D-FFT application

The 2D-FFT is used in many applications and is often considered representative of workloads that operate on matrices that are distributed across the nodes of a parallel machine[10]. The 2D-FFT algorithm is composed of the following four steps.

- Compute the 1D-FFT for each row (fftw_comp_nd)
- Transpose the matrix (redistribution of data) (fftw_transpose(1))
- Compute the 1D-FFT for each row (fftw_comp)
- Transpose the matrix (redistribution of data) (fftw_transpose(2))

Each of these steps were directly translated into Coven modules and these steps iterated thrice in order to compute three FFTs. The pseudo Coven code for the application is shown in figure 3.6

N-Body application

The N-Body application is a program which models point-masses in three dimensional space and how they interact due to gravitational forces. There are N individual

point-masses (bodies) which are divided between the parallel tasks. The N-Body application is iterative, moving forward in time steps. In the most complete case of N-Body, for each iteration each body must compute the forces applied on it by each other body. In a parallel system this requires that each body eventually reach each parallel task. Other N-Body algorithms, such as Barnes Hut, attempt to improve on the performance by assuming that bodies far away have no force affect on each other. With this assumption, it is not necessary for each body's forces to be computed with each other body's forces and, therefore, not all bodies will need to be transported to each parallel task. For this example, the application was implemented in such a way that the bodies were divided into groups and those groups were passed around in a ring until each parallel task had received them. After receiving a portion of the bodies, each task computes the partial forces those bodies apply to the bodies it is responsible for. Only after every task has computed all partial forces does the application have a global solution for the current time step. Time then advances to the next iteration[10]. The pseudo Coven code for this application is show in figure 3.7

Using these two example applications we shall now explain the various visualizations that are provided by the Coven profiler.

Total Module Runtime View

This view displays the *total execution time* of a Coven module during the lifetime of a parallel program. Here the total execution time refers to the sum of the execution time of the Coven module on all the nodes of the cluster. This view can enable a Coven programmer to compare the execution times of the different Coven modules that are used to develop the parallel program. Figure 3.8 shows this view for the 2D-FFT application. As seen in the figure, this visualization is a bar graph where the x-axis represents the *Coven modules* while the y-axis represents the *total execution*

```
. . .
   thread_master nbody_generate_planar(args)
   thread_master nbody_scatter_multiple(args)
      PARALLEL {
          for(i=0;i<iterations;i++){
             thread_slave nbody_copy_buffer(args)
             for(j=0;j<num_of_slave_procs;j++){
               thread_slave nbody_compute_forces(args)
               thread_slave nbody_parallel_shift(args)
             }
             thread_slave nbody_compute_positions(args)
             thread_slave nbody_visualize(args)
          }
     }
   thread_master nbody_gather_multiple(args)
. . .
```

Figure 3.7: Pseudo Coven code for N-Body application

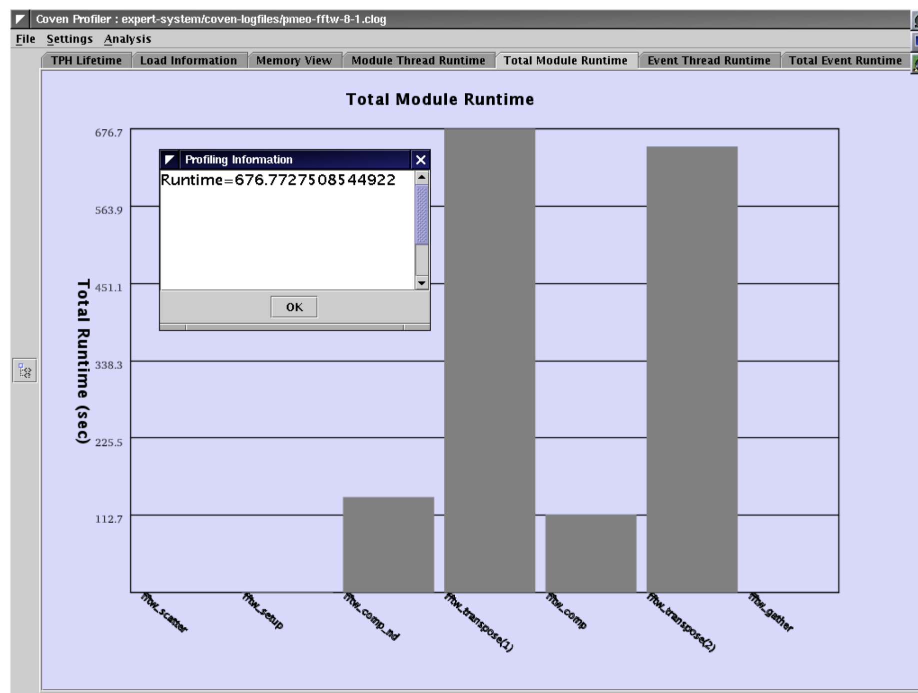

Figure 3.8: Total Module Runtime

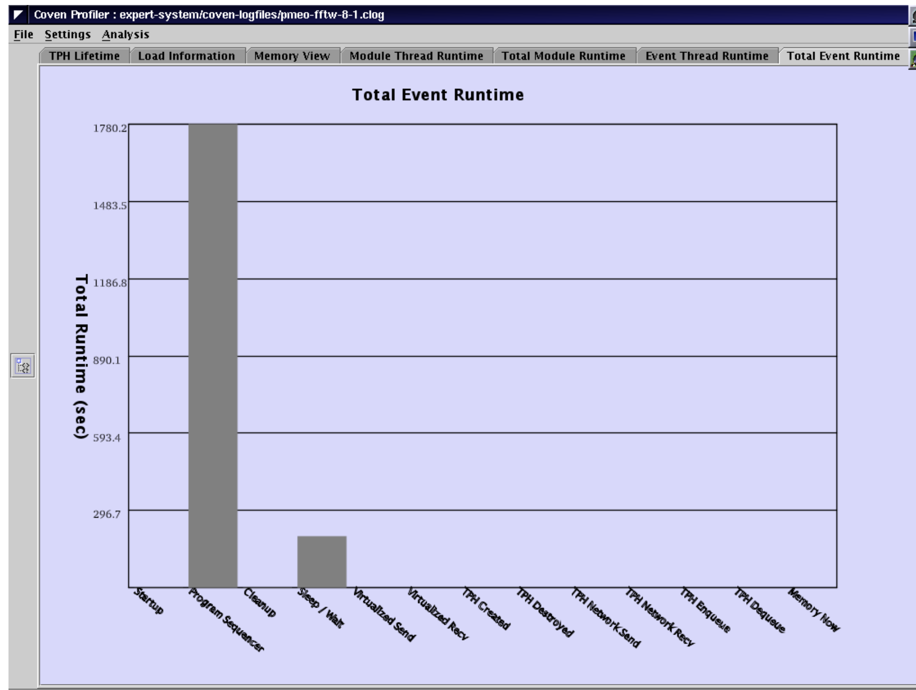

Figure 3.9: Total Event Runtime

time of these modules in seconds. For example, the fftw_transpose(1) module takes a total of 676.8 seconds to execute. This value is obtained by adding the execution time of this module on all the nodes of cluster. The dialog box exhibiting this value is displayed by right clicking the bar corresponding to fftw_transpose(1) module in the bar graph.

Total Event Runtime View

This view is similar to the Total Module Runtime View except that instead of displaying the total runtime for the Coven modules, it displays the total runtime for all of the *Coven auxiliary events*. Figure 3.9 shows this view for the 2D-FFT application.

Module Thread Runtime View

This view displays the time taken by a module to execute in each Coven thread. Figure 3.10 shows this view for the 2D-FFT application. As seen in the figure, this

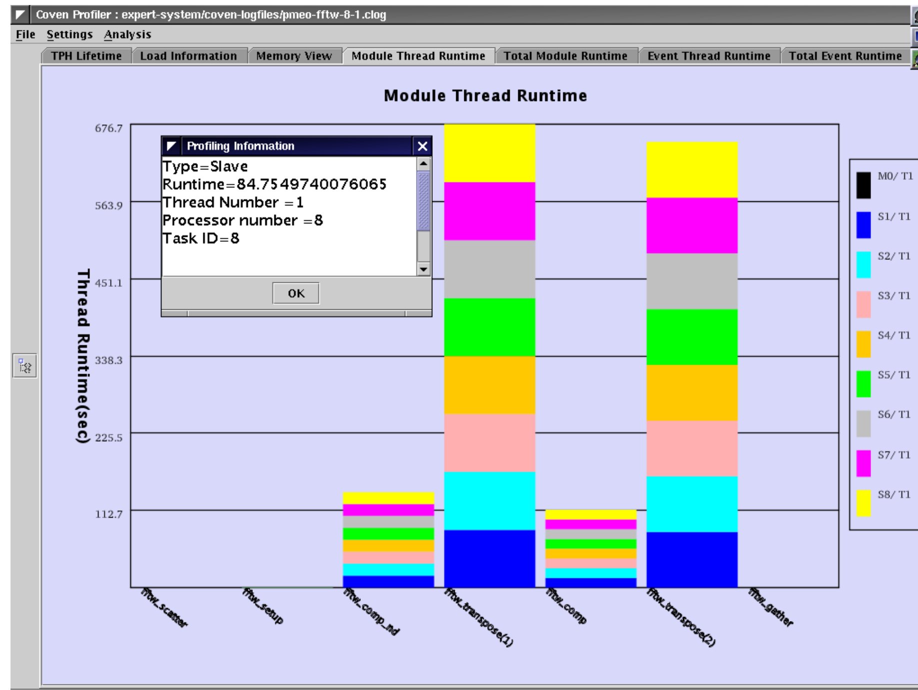

Figure 3.10: Module Thread Runtime

visualization is a stripped bar graph where the x-axis represents the *Coven modules* while the y-axis represents the *thread execution time* of these modules in seconds. The strips in the bar graph correspond to the different Coven threads in which the module runs. A legend which maps the Coven threads to the strip colors is provided at the right side of the graph. The height of a strip corresponds to the execution time on the module in a particular thread. For example, the fftw_transpose(1) module took 84.75 seconds to execute on thread 1 of the slave processor 8. The dialog box displaying these values can be obtained by right clicking the corresponding strip on the bar graph. In addition to displaying the thread runtime information for the module, the dialog box also displays the following data

- The type of processor (Master/Slave) on which the module was executed.
- The processor identifier.
- The thread identifier.

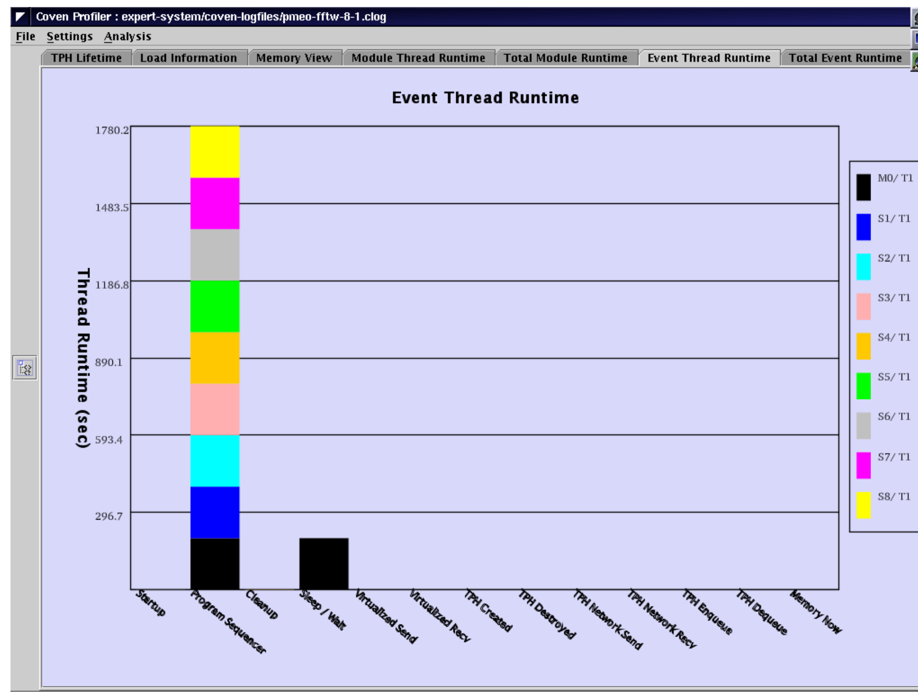

Figure 3.11: Event Thread Runtime

- The MPI parallel task identifier.

Event Thread Runtime View

This view is similar to the Module Thread Runtime View except that instead of displaying the thread runtime for the Coven modules, it displays the thread runtime for all of the *Coven auxiliary events*. Figure 3.11 shows this view for the 2D-FFT application.

Load Information View

This view is used to study the load that is exerted by the Coven modules during the lifetime of the parallel program. This view is basically a GANTT chart with the *time* in seconds on the x-axis and the *Coven program threads* on the y-axis. Figure 3.12 shows this visualization for the 2D-FFT program. The colored rectangular blocks across a Coven thread represent the sequential execution of the Coven modules in

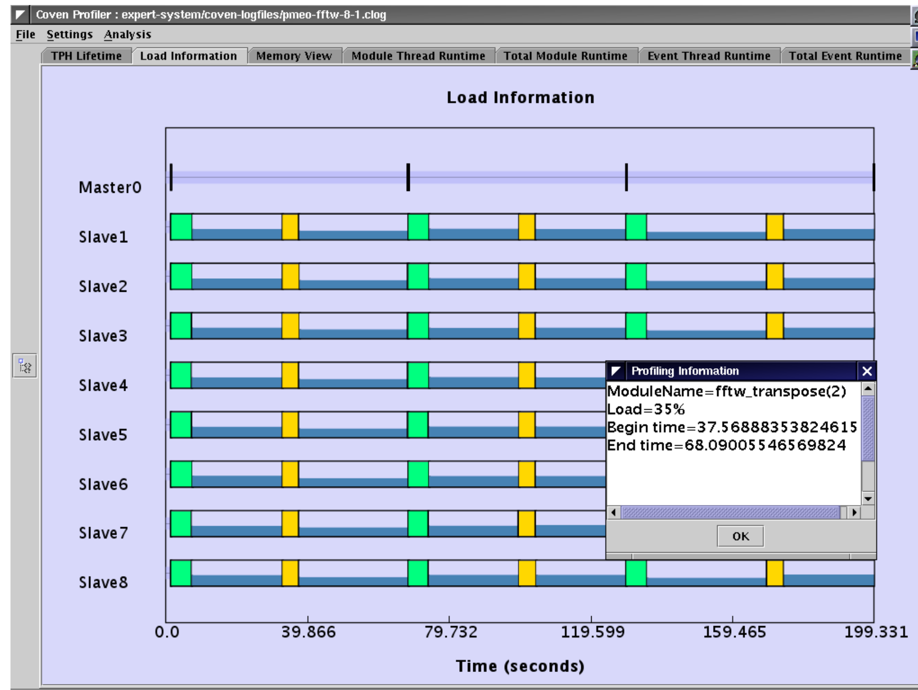

Figure 3.12: Load Information

that particular thread. The length of the colored rectangular block corresponds to the amount of time in seconds for which the module runs while the amount of filled region corresponds to the percentage of load exerted by the Coven module. For example, as seen from the dialog box in the figure the amount of load exerted by the fftw_transpose(2) module is 35%. This information box is obtained by right clicking on any of the rectangular box corresponding to the fftw_transpose(2) module on the graph. In addition to the load, the dialog box also displays the start and the end times of the module execution.

Memory Consumption View

This view enables a Coven programmer to study the memory patterns of the different Coven program threads during the lifetime of the parallel application. Figure 3.13 shows this visualization for the N-Body problem. As seen in the figure, this view is a graph with the time in seconds on the x-axis and the amount of memory consumed in

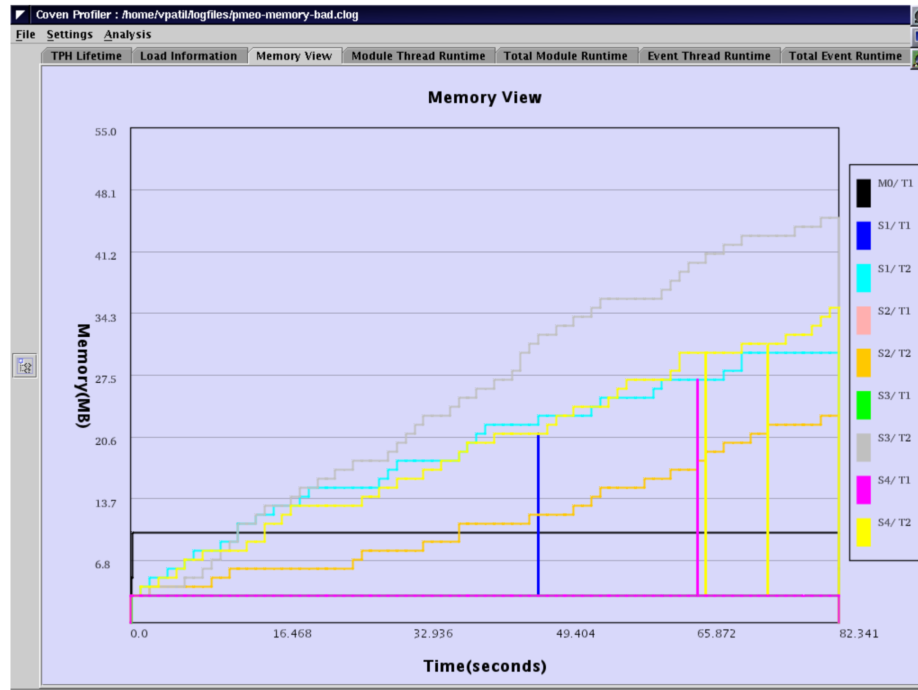

Figure 3.13: Memory Information

mega bytes on the y-axis. The colored lines in the graph correspond to the memory consumption by the different Coven threads. A legend mapping the colored lines to Coven threads is provided on the right hand side of the graph.

TPH Lifetime View

The TPH Lifetime view is the most important view and is used to study the flow of TPHs (Tagged Partition Handles) [10] during the lifetime of the Coven program. This view is basically used to study how the Coven architecture works and was specifically developed for advanced programmers that are involved in the development of the software framework. As seen in figure 3.14, the visualization is basically a GANTT chart with the *time* in seconds on the x-axis and the *Coven program threads* on the y-axis. The colored rectangles represent the modules through which the TPHs flow. Each of these colored rectangle has a number as well as a thin colored strip at the bottom to visually identify the TPH following through the module. The lines

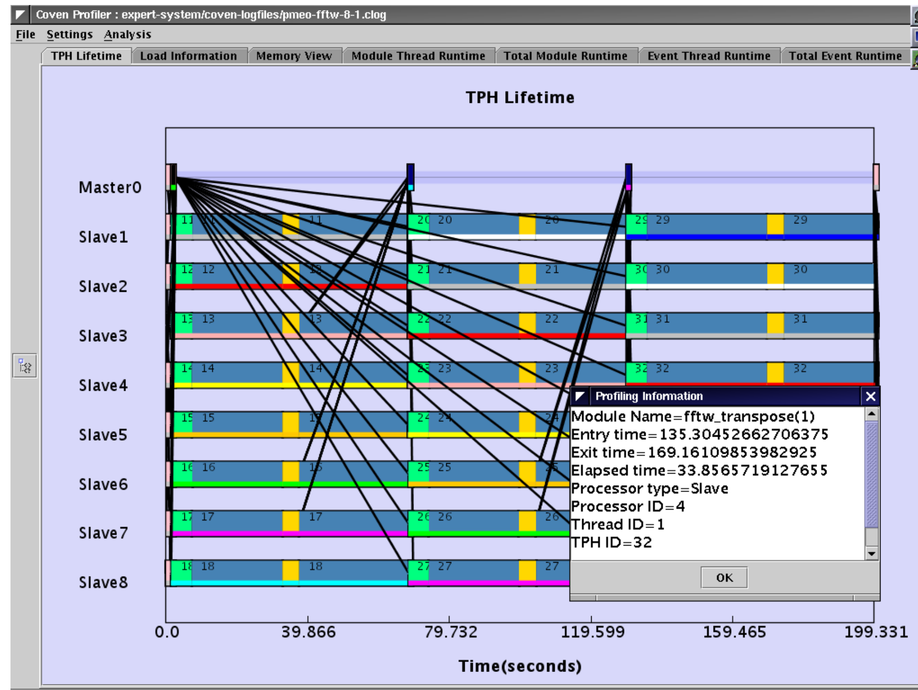

Figure 3.14: TPH Lifetime View

connecting the rectangles represent the flow of TPHs from one module to another. As in the load view, it is possible to obtain information regarding the TPH flowing through a module by right clicking the the rectangle corresponding to the module. This pops up an information box which displays the following data

- Module name
- Entry time of the TPH in module
- Exit time of the TPH from the module
- The amount of time spent by the TPH inside the module
- The processor type (Master/Slave)
- The processor identifier
- The thread identifier
- TPH identifier

3.3 Rule Based Expert Agent

The Coven profiler offers the programmer with several graphical views that represent the behavior of the parallel program. From these graphical views the Coven programmer needs to answer questions like : How does the application perform? Where in the execution does the performance fall? What causes the performance fall and how can the performance be improved? When the programmer is trying to answer the above questions, he must face the problem of handling enormous amount of graphical information, selecting views which explain the sensitive aspects of performance, understanding the views to find which performance problems do actually appear and relating them with their ultimate causes in the program code. Hence the programmers require a very high level of experience in order to derive any conclusions regarding the program behavior using the visualization tools[7].

It is also important to note that a system which can provide higher level of performance measurement and analysis is more helpful in the performance tuning of parallel program. For example, whether the programmer adopts a proper program strategy or algorithm is one of the most important factors which affect the performance of parallel programs. Therefore, a helpful performance tuning tool should be able to assist programmers to optimize the strategy or algorithm in their parallel programs. Providing higher level performance measurement and analysis is highly desirable [15].

Keeping these objectives in mind a rule based expert system has been developed for automated post mortem performance analysis of the parallel programs written in Coven. The expert agent has been built using JESS (Java Expert System Shell)[9]. The heuristics used to detect the problems in the parallel program are encoded in the form of JESS rules. The syntax of these rules is similar to the LISP language syntax. The rules are executed by the expert system shell coupled with the information extracted from the log file and necessary suggestions are then made to the programmer. Figure 3.15 shows an example of a JESS rule which suggests multithreading when a

```
(defrule MultiThreading
         (and (number-of-threads 1)
            (is-coven-module ?m1)
            (is-coven-module ?m2)
            (or (has-low-load ?m1 ?threadid)
                (has-low-load ?m2 ?threadid))
            (or (follows ?m1 ?m2 ?processorid ?threadid)
               (follows ?m2 ?m1 ?processorid ?threadid))
            (not (place-on-seperate-threads ?m1 ?m2))
         )
   =>
   (assert (place-on-seperate-threads ?m1 ?m2))
)
```

Figure 3.15: A sample JESS rule

---

single threaded Coven program has modules exhibiting low load load and which run sequentially on the same thread.

Depending on the parallel program run, the expert agent may make one of the following suggestions

- Enable multithreading
- Enable load balancing
- Enable shared memory

The heuristics behind each of these suggestions is explained in the following sections.

Enable multithreading

In parallel computing applications, there are periods of execution when the data required for computing needs to be accessed outside the CPU. For example, when a task does disk I/O or network I/O the CPU remains ideal during the data access time. A common approach to overcome this latency is to use a multitasking operating

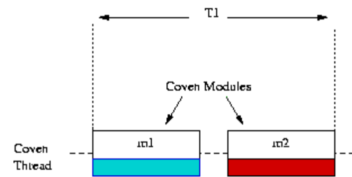

Figure 3.16: Before Multi-threading

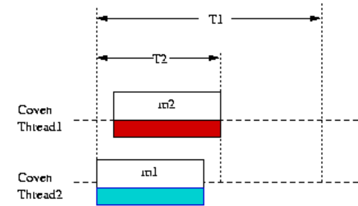

Figure 3.17: After multi-threading

system and scheduling running processes concurrently. The idea being that while one process is waiting for the data, it gets scheduled out while another process is scheduled in thereby increasing the overall CPU utilization. This is generally termed as *multi-threading* or *asynchronous programming*[10].

Since the Coven model separates the application code from the application state, it makes it simple to schedule tasks to execute in parallel. TPHs are moved around to combine with modules to form a task, and this task can execute anywhere in the system at any time. In the same manner, the Coven multi-threading implementation utilizes multiple threads of control concurrently executing on the same parallel node. These threads can be assigned any module and as TPHs arrive they form tasks which execute concurrently with other threads[10].

A typical scenario where multithreading can prove beneficial to Coven program execution is shown in figure 3.16. This figure can be thought of as the load view, where a rectangle represents the Coven module execution and the amount of filled area corresponds to the percentage load exerted by it. Figure 3.16 shows a Coven program with single thread of control on each node of the cluster. This thread executes two Coven modules *m1* and *m2* sequentially and both of which execute a low load ($\leq$ 50%) on the CPU node. If there is no data dependency between the two modules it might be possible to efficiently utilize the CPU by placing the modules on separate threads and thereby decrease the overall runtime as shown in figure 3.17.

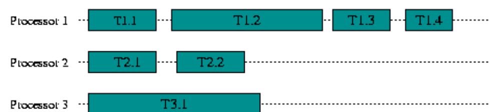

Figure 3.18: Without load balancing

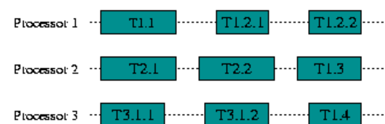

Figure 3.19: With load balancing

Enable load balancing

There are several parallel computing applications in which the workload divided among the parallel tasks is variable. Examples of such applications include ray tracing, fractal generation and so forth. In these type of applications it is extremely difficult, if not impossible to determine the amount of computation to be performed by each parallel task. As a result of which some of the nodes remain idle while the others keep crunching numbers for a long time. This problem of uneven processor utilization is referred to as the load balancing problem and is formally classified as an NP-complete optimization problem[12].

Recall that the basic unit of work in Coven is a TPH. Each parallel Coven thread operates on a single TPH at a time, while maintaining queues of TPHs to be processed and to be sent. Coven tries to implicitly solve the load balancing problem by implementing the *Random Stealing* dynamic load balancing algorithm. While the random stealing algorithm is useful in a large class of applications, iterative applications that process a set of data over and over again have a more difficult time taking advantage of this algorithm. Figure 3.18 shows the TPH view for a parallel application with load imbalance. Such applications need to be explicitly load balanced by partitioning the data into more TPHs than there are processors and by inserting a dynamic load balancing system module provided by Coven, into the data flow graph of the parallel program at the end of an iteration[10]. Once the parallel application has been explicitly load balanced the TPHs get redistributed as shown in figure 3.19 thereby optimizing the processor utilization.

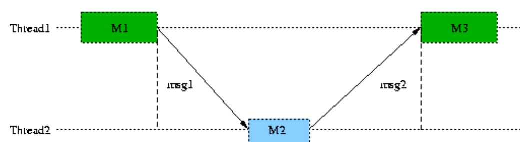

Figure 3.20: Without shared memory, Coven uses MPI for inter-thread messages

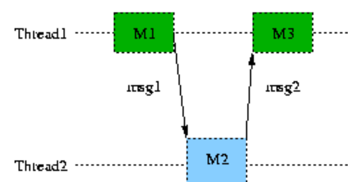

Figure 3.21: With shared memory

The expert system tries to detect the imbalance in workloads by first determining the efficiency of each Coven thread. The efficiency of a Coven thread is defined as the percentage of time the thread is busy executing out of the total parallel runtime. The expert system then compares the efficiency of each Coven thread with the efficiency of every other thread and when it notices that the difference in efficiencies of two or more threads is greater that a fixed threshold (35%) it suggests the user to explicitly load balance the parallel application.

Enable shared memory

Coven offers a programmer an option to switch on shared memory to transfer data transparently between the Coven threads running on the same processor. This adds a small amount of overhead since Coven internally performs all the shared memory management. If shared memory is switched off, the shared data is exchanged between the Coven threads using normal MPI routines and this results in poor performance when the size of the data being exchanged is large. Experimental results have shown that it requires 7 seconds to transfer 256 MB of data among Coven threads running on the same processor when the shared memory option is switched off[10]. When the shared memory feature is switched on, it requires less than 0.05 seconds to transfer the same amount of data.

The expert system tries to determine whether enabling the shared memory feature can benefit a parallel application by studying the pattern of the inter thread messages. It computes the average of the inter thread message transfer times and if it is greater

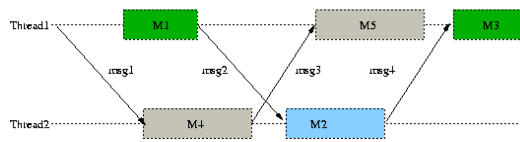

Figure 3.22: Overlapping message transfers with module execution, without shared memory

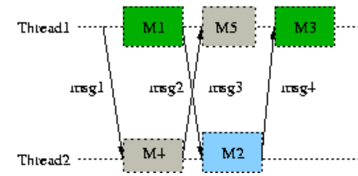

Figure 3.23: Overlapping message transfers with module execution, with shared memory

than a fixed threshold (5% of the program runtime) it suggests switching on the shared memory feature. Initially only those inter thread messages were considered for which the message transfer time did not overlap with the module execution time on the destination thread. For example, as shown in figure 3.20 when message msg1 is transferred from thread1 to thread2, the message transfer time does not overlap with module execution time on thread2 and hence this message was considered as a good candidate for determining whether switching on shared memory feature would be beneficial. The transfer time of all such messages was used to determine the average inter thread message time and if this value was greater that 5% of the program runtime, switching on the shared memory feature was suggested. For applications having this type of message pattern, once the shared memory feature was switched on the inter-thread message transfer time was expected to decrease considerably (ideally 0) as shown in figure 3.21.

While making the decision regarding shared memory, the aforementioned heuristic ignored the messages in which the transfer time of the message overlapped with the module execution time on the destination thread. However consider an inter thread communication pattern as shown in figure 3.22. While msg2 is transfered from thread1 to thread2 module M4 executes on thread2 and thus the communication time of the message is overlapped with the execution time of the module. Similar is the case with message msg4. The earlier heuristic did not consider these messages while making decisions related to shared memory since it assumed that there was work

been done on the destination thread while the message is being transfered. However it is important to note that the execution time of a module not only includes it's computation time (actual work) but it also includes the time for which the module waits in order to receive the inter thread messages. Hence an additional performance gain can be obtained by speeding up module execution by decreasing the inter thread message transfer time using shared memory thereby decreasing the overall runtime of the parallel application as shown in figure 3.23. Therefore a new heuristic has been defined which considers all of the inter thread messages and if the average time required to transfer these messages is greater than 5% of the program runtime, it suggests enabling the shared memory feature in Coven.

The above three heuristics have been encoded in form of JESS rules in a text based script file. These rules require some parameters which are extracted from the CLOG file using the data model. The rules coupled with the extracted information are then executed by the rule engine and an appropriate decision is made with the goal of improving the performance of the parallel application if possible.

Chapter 4

Case Studies

The chief goal of this research is to develop performance analysis tools that enable a programmer to easily identify the common problems that are encountered while developing parallel applications using Coven. This chapter discusses several case studies that are used to demonstrate how the Coven profiling architecture facilitates the study of the different aspects of a parallel program in order to identify the performance bottlenecks. In these case studies different parallel algorithms and one synthetic application have been implemented. In addition to visualizing the runtime information using the Coven profiler to identify performance problems, the case studies are used to demonstrate how the Coven expert system provides a parallel programmer with suggestions to enhance a parallel application, in order to improve it's performance.

4.1 Work Environment

The case studies were run on a 76 node Beowulf cluster consisting of nodes with two 1GHz Pentium III processors, 1GB of RAM, and connected by Fast Ethernet. Each node of the cluster ran Linux kernel 2.6, and MPICH2 0.93. All of the Coven programs were compiled with GCC version 3.3.2 using maximum optimizations.

```
. . .
        PARALLEL {
            thread_slave1 CMM_init_block(args)
            thread_slave1 CMM_read_mat1(args)
            thread_slave1 CMM_read_mat2(args)
            thread_slave1 CMM_gen_zero_mat(args)
            thread_slave1 CMM_get_shifts(args)

            for(__phase = 1 ; __phase<= phaseCount;
                               __phase += 1)
            {
                thread_slave2 CMM_shift_mats(args)
                thread_slave2 CMM_mat_mult(args)
            }
        }
. . .
```

Figure 4.1: Cannon's matrix multiplication algorithm

The following two sections shall now discuss experiments that illustrate how the visualizations provided by the Coven profiler help in identifying common performance issues.

4.2 Comparing module run times

Many times it is difficult to determine the amount of time a parallel program spends doing a segment of code. This can make optimization difficult in that it is sometimes not clear where performance is being lost, and where additional performance can be gained.

In this case study we look at a Coven application which is an implementation of the **Cannon's Matrix Multiplication** algorithm [19], with poorly designed network communication module which performs unnecessary computations. The steps involved in this algorithm are shown in figure 4.1 in the form of Coven pseudo code. The modules CMM_shift_mats and CMM_mat_mult perform most of the operations

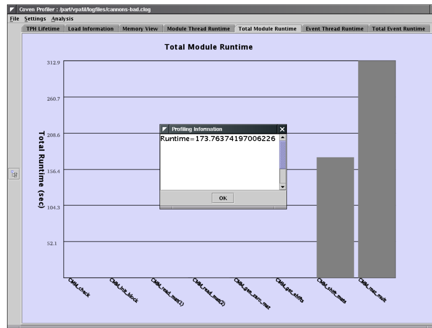

Figure 4.2: Module Runtime before optimization

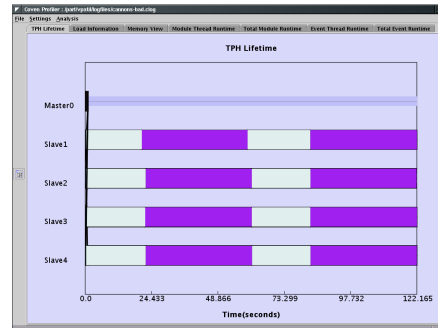

Figure 4.3: TPH view before optimization

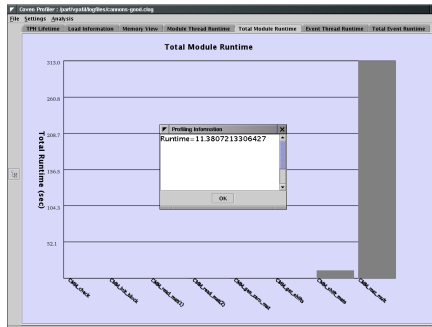

Figure 4.4: Module Runtime after optimization

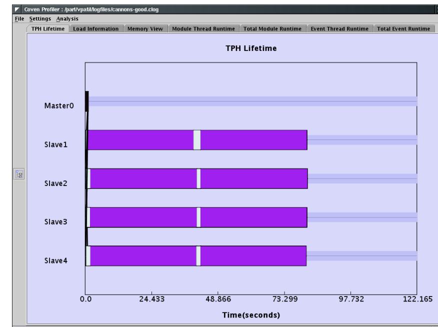

Figure 4.5: TPH view after optimization

of the parallel program. The CMM_shift_mats is basically a communication module which transfers the block data between processors while the CMM_mat_mult module performs the actual multiplication of the blocks of data available.

The parallel application was executed to multiply two 2400x2400 matrices and the log file generated was then visualized using the Coven profiler. As seen from the views in figure 4.2 and 4.3 the CMM_shift_mats took a considerable amount of the program time even though it was simply responsible to move the blocks of data among the appropriate neighbors. Focus was then placed on improving this module by removing the unnecessary computations that were being performed. Figures 4.4 and 4.5 show the resulting profiler views after the CMM_shift_mats module was optimized. As seen from the figure the total runtime of the communication module dropped from 173.76 sec to 11.38 sec while the total application runtime decreased from 122.16 sec to 81.76 sec.

4.3 Detecting memory problems

Memory leaks can be hard to detect and even tougher to locate in complex parallel applications. Coven takes a snapshot of memory usage at many points during program execution using the Linux /proc file system as explained in section 3.1.2. This can be helpful to identify the memory leaks and pinpoint their exact location. The following case study demonstrates a Coven program with memory leak and how the profiler was used to identify and locate the problem. In this case study we used the N-body application explained in section 3.2.2, with 10,000 bodies being modeled. An initial Coven implementation of this program exhibited performance problems in that the amount of time each iteration took began to slow down at a high number of iterations. The Coven profiler was then used to study the memory usage of the program. A screen shot showing this view appears in Figure 4.6. As seen from the figure, the

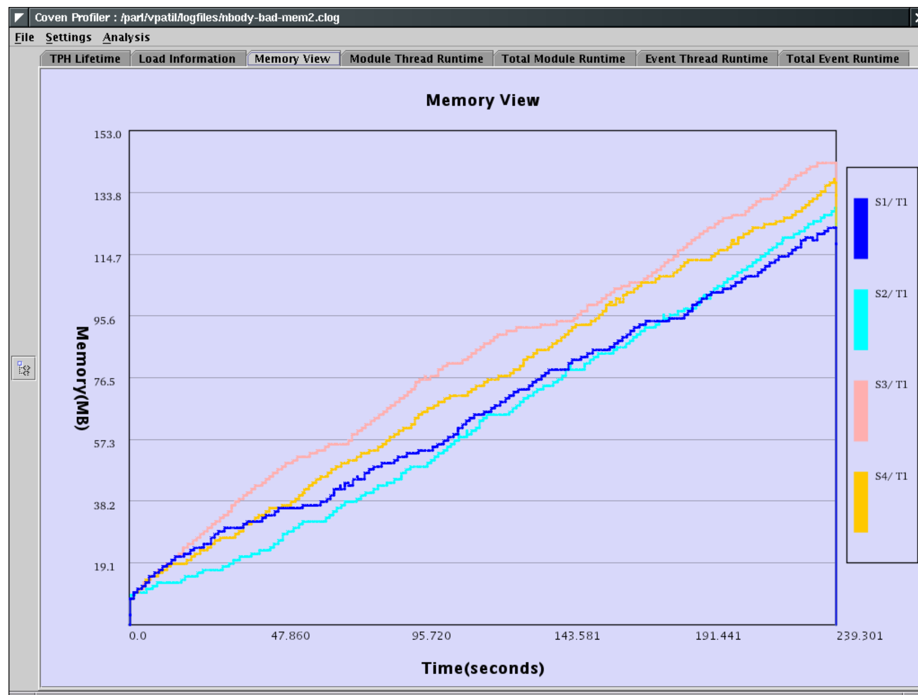

Figure 4.6: N-body application with memory leak

Figure 4.7: N-body application without memory leak

amount of memory being used grew progressively with time and this was causing the application to slow down. The different Coven modules that were employed to construct the parallel program and which allocated memory dynamically were then studied. This led to the identification of the offending module (nbody_parallel_shift) which dynamically allocated memory without ever releasing it. Upon fixing the error, the memory ceased to grow every few iterations as shown in figure 4.7 and the program began to perform as expected.

4.4 Automatic Performance Analysis

The case studies presented in the following sections illustrate how the rule based expert agent system is able to provide suggestions to the parallel programmer in order to improve the runtime of the parallel applications.

4.4.1 Enabling Multithreading

As explained in section 3.3, multithreading can benefit a Coven program when the modules that run on the same Coven thread exhibit a low load and are sequentially executed one after the other. To illustrate this case a 10,000 x 10,000 2D-FFT application was implemented in Coven and executed over 9 processors. The execution steps of this application are explained in detail in section 3.2.2. The algorithm was first implemented in single threaded mode. As seen from the load view in figure 4.8, some of the Coven modules exhibited a low load ($\leq$ 50%) during execution. The expert system was then consulted for any possible suggestions to improve the runtime of the program. The expert system could detect that the modules having low load ran sequentially on the same Coven thread and hence it suggested multithreading the application as seen in figure 4.8. The application was then accordingly multi-threaded with the *fftw_compute* and *fftw_transpose* modules being placed on separate threads

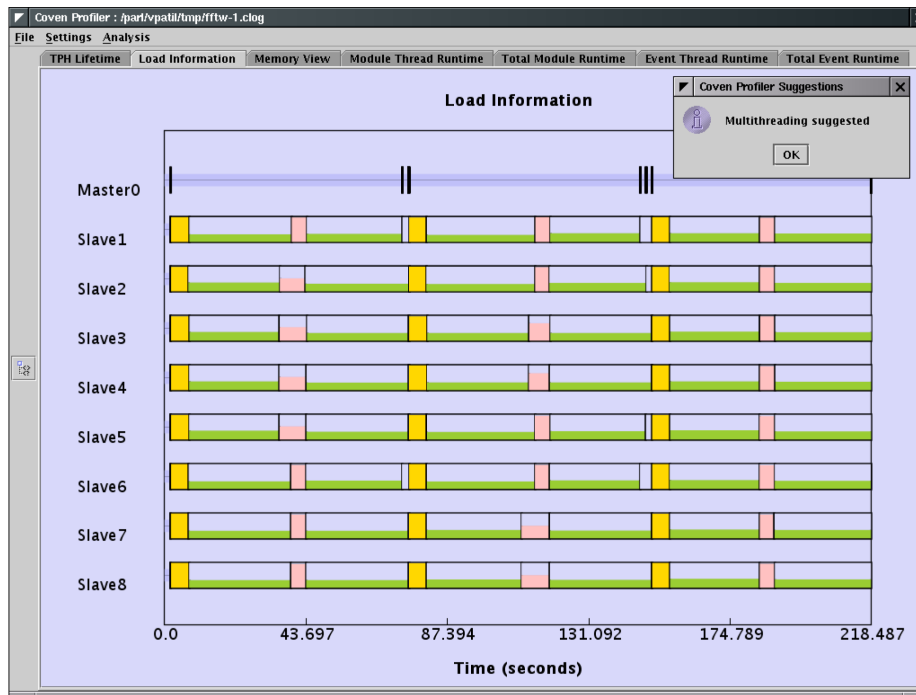

Figure 4.8: Before Multithreading

Figure 4.9: After Multithreading

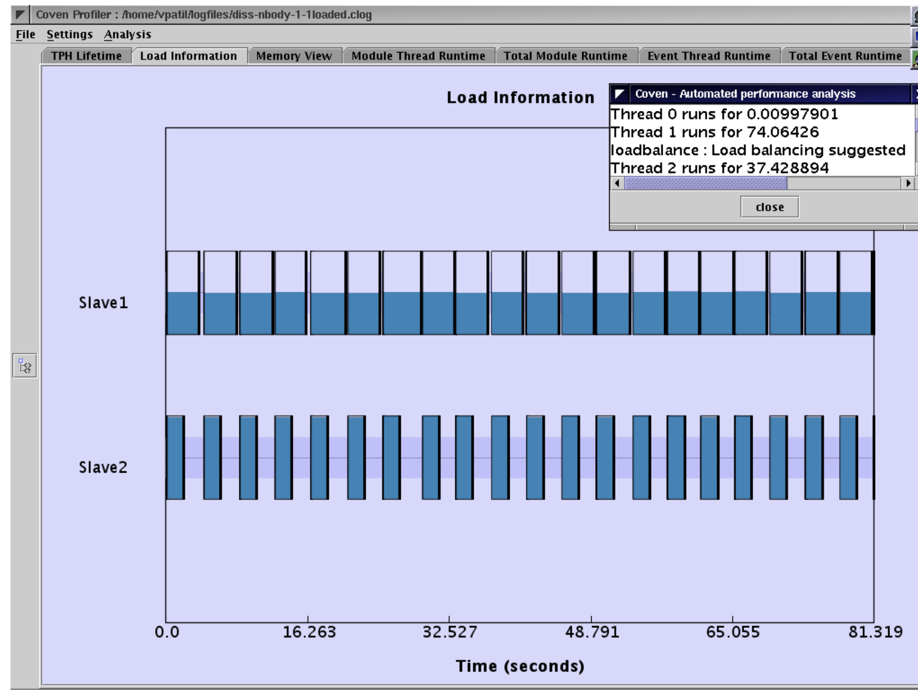

Figure 4.10: Before Load Balancing

as a result of which the runtime of the application dropped from 218.5 seconds to 193.7 seconds as seen in figure 4.9.

4.4.2 Enabling Load Balancing

There are two Dynamic Load Balancing (DLB) optimizations available in Coven. The first DLB algorithm transparently moves TPHs between parallel tasks as less loaded tasks run out of TPHs to process. Coven currently implements this functionality using the Random Stealing Algorithm. However there are some iterative applications that process a set of data over and over again, which cannot directly benefit from Coven's random stealing DLB algorithm. Applications of this type need to be load balanced explicitly. This less transparent DLB scheme requires breaking of data into smaller chunks and inserting of a, pre-built DLB system module into the data flow graph of the user application at the end of the iteration[10]. The following case study illustrates a Coven program which does not benefit from the implicit load balancing

scheme implemented in Coven. The expert agent is then used to detect this problem and suggest the user to explicitly load balance the parallel program using the system module provided by Coven.

To illustrate this case, we consider the N-Body application explained in section 3.2.2 with 10,000 bodies. The application was executed on three nodes with one master node and two slave nodes. The N-Body application is perfectly balanced because there are a predefined number of calculations that are required to compute the new force applied to each body. In an effort to experiment with load imbalances with this type of application we introduced external load. We define external load as some force other than the application which causes the application to execute with an imbalance. To demonstrate the need for load balancing under these conditions, the N-Body application was executed while one of the slave nodes was heavily loaded with another application. This other application caused the Coven parallel task to only get access to the CPU about 50% of the time. Therefore, this portion ran half as fast as the other parallel task. Figure 4.10 is a screen-shot of the Coven profiler which depicts this application. The first process (Slave 1) is the one competing for CPU. Slave 2 can be seen to be idle for a large portion of the runtime, due to delays waiting on the other parallel task. Also as seen from the dialog box thread 1 runs for 74.06 seconds while thread 2 runs for 37.42 seconds. Thus the difference in the efficiency of the two threads is 45.04% ($\geq$ 35%). This imbalance in load is detected by the expert agent using the heuristic explained in section 3.3 and it then suggests explicitly load balancing the application.

Figure 4.11 shows the profiler screen-shot of the parallel application after it has been load balanced. The application was explicitly load balanced by adding the load balancing system module and by partitioning the data into more pieces than there are parallel nodes. As a result of this not only did the runtime of the application

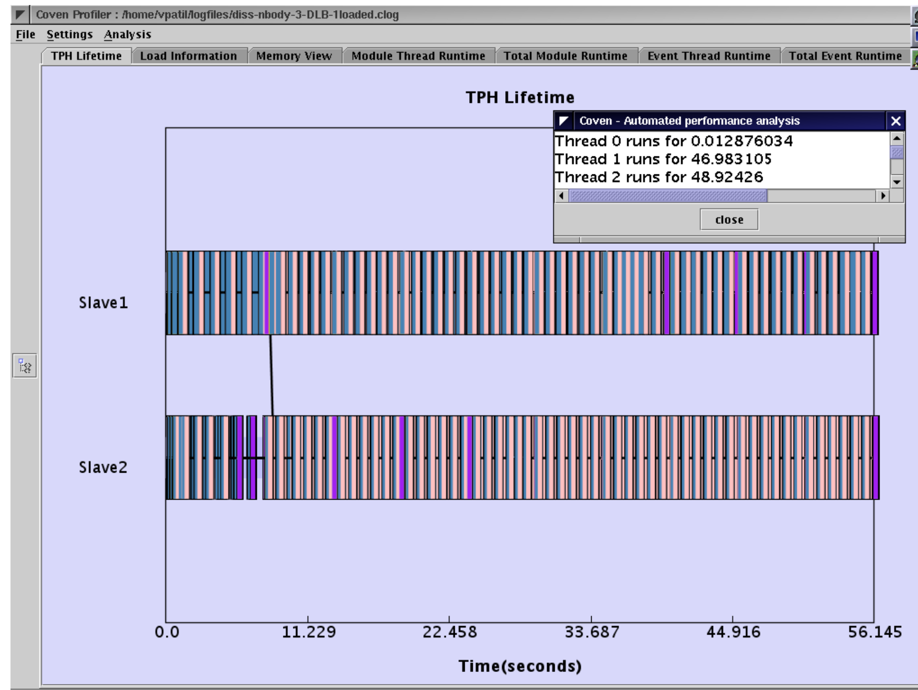

Figure 4.11: After Load Balancing

decrease by 31% but the parallel tasks reached a balance between their individual run times.

4.4.3 Enabling Shared Memory

TPHs are transferred between Coven Threads using MPI. However Coven provides an option to transfer TPHs among the program threads running on the same processor using shared memory. From a user's perspective it is essentially transparent how Coven handles the movement of the TPHs among the different program threads. In order to enable the shared memory feature, the user needs to simply set the value for maximum size of the shared memory region. There is certainly a trade off when using the shared memory system as it requires synchronization among the Coven threads running on the same processor. Also the shared memory feature is only beneficial in the case when the amount of data being exchanged between the threads running on the same processor is large[10].

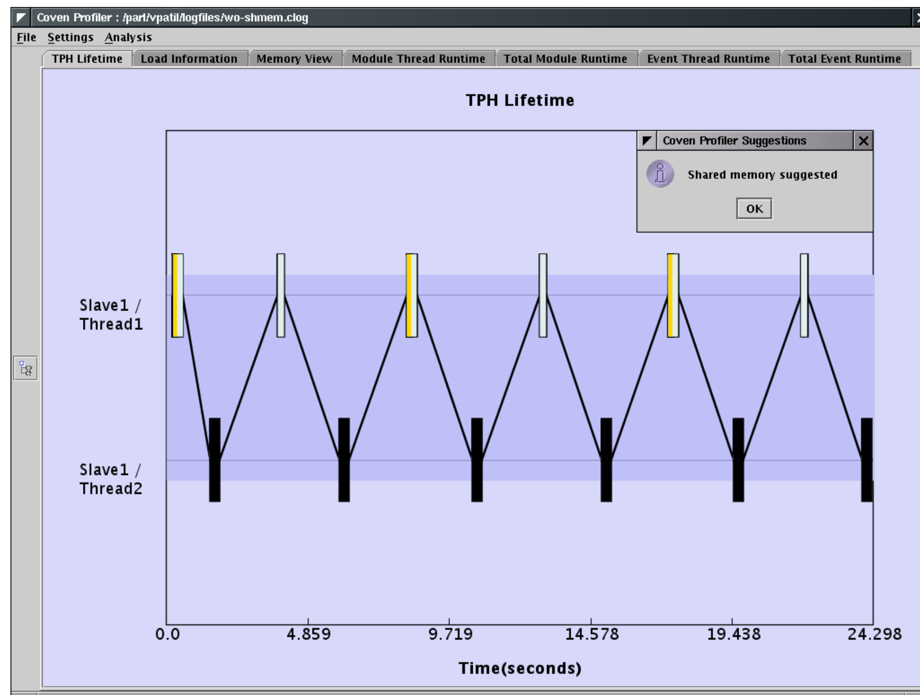

Figure 4.12: Before Shared Memory

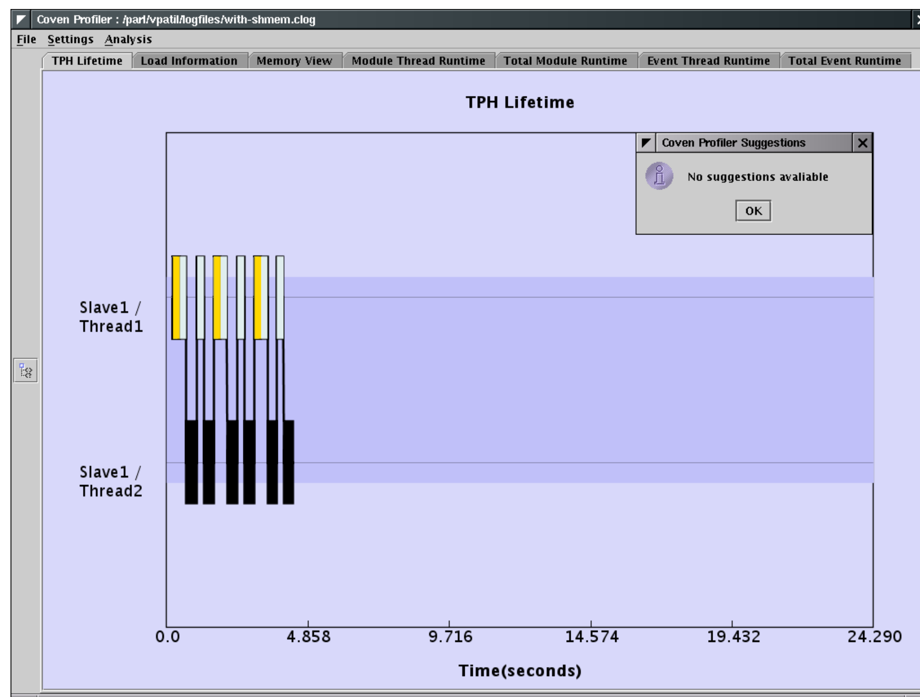

Figure 4.13: After Shared Memory

The following case studies demonstrate how the expert system tries to detect when using the shared memory feature would prove beneficial to a multi threaded Coven application.

Synthetic Coven Application

This case study consists of a synthetic Coven application in which modules running on separate Coven threads (on the same processor) simply exchange 384 MB of data. The multi threaded application was executed on 5 nodes in single processor mode, with one master node and four salve nodes. Figure 4.12 shows the profiler screen-shot for this application with the shared memory feature switched off. For simplicity of visualization, the screen shot displays information regarding the threads running on a single node (slave1). As seen from the figure the inter thread messages take a while to transfer between the threads running on the same processor (average of 1.63 sec, $> 5\%$ of the parallel program runtime). Also during the transfer of these inter thread messages there was no useful computation done on the destination thread and it remained ideal during the message transfer time. This pattern was recognized by the expert agent and it suggested switching on the shared memory feature in Coven. Figure 4.13 shows the profiler screen-shot for the same application with shared memory enabled. As seen from the figure the inter thread message transfer time was almost negligible as a result of which the program runtime decreased from 24.29 seconds to 4.43 seconds.

2D-FFT Application

This case study consists of the multi-threaded FFTW Coven application explained in section 3.2.2. This application was executed to compute a 10,000 x 10,000 two dimensional FFT and was run on 9 nodes with 1 master node and 8 slave nodes. Figure 4.14 shows the TPH view for the application execution with shared memory

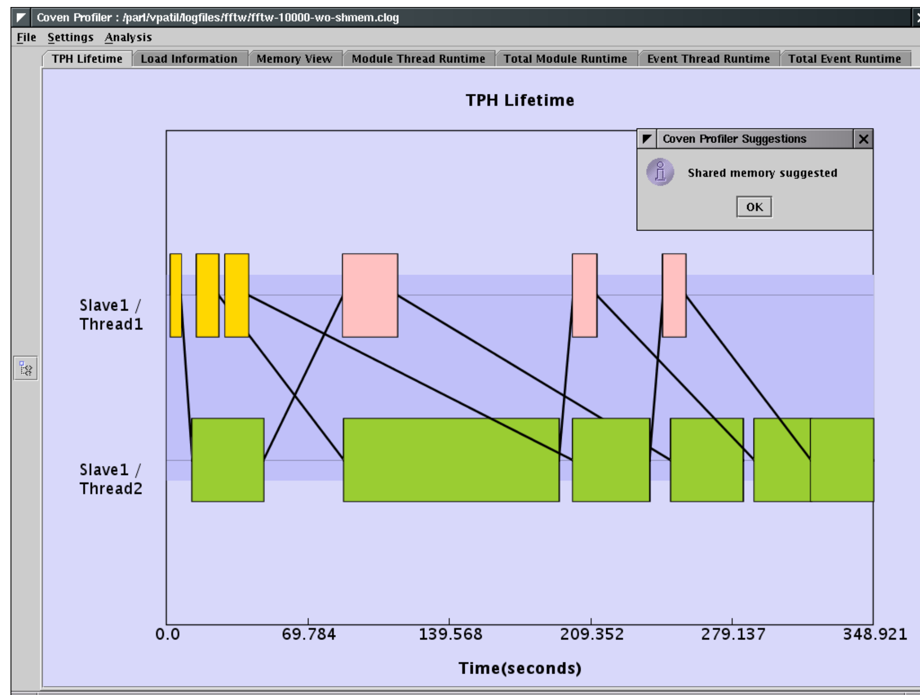

Figure 4.14: TPH view for FFTW application without shared memory

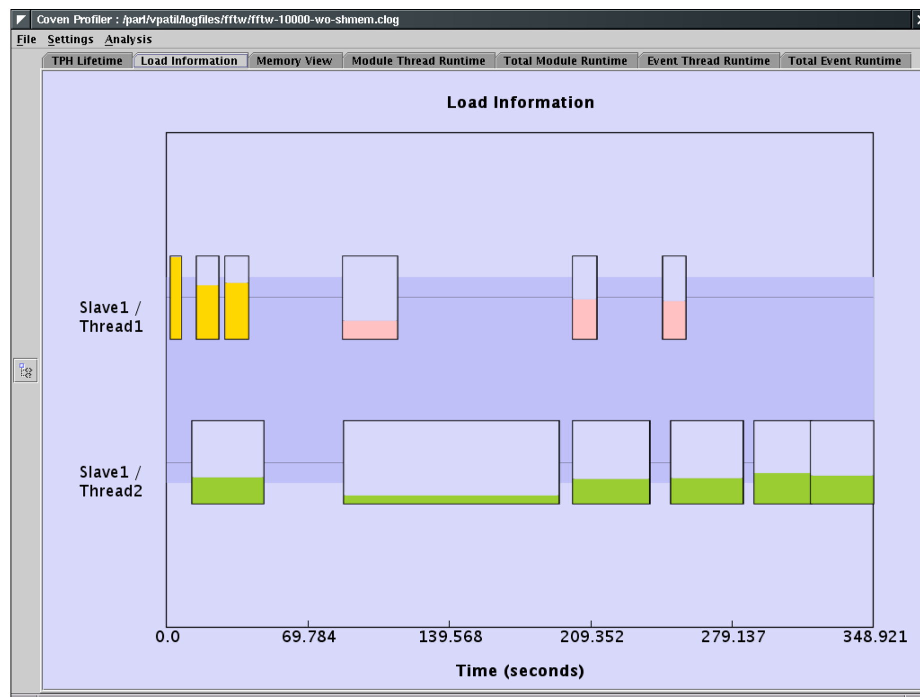

Figure 4.15: Load view for FFTW application with shared memory

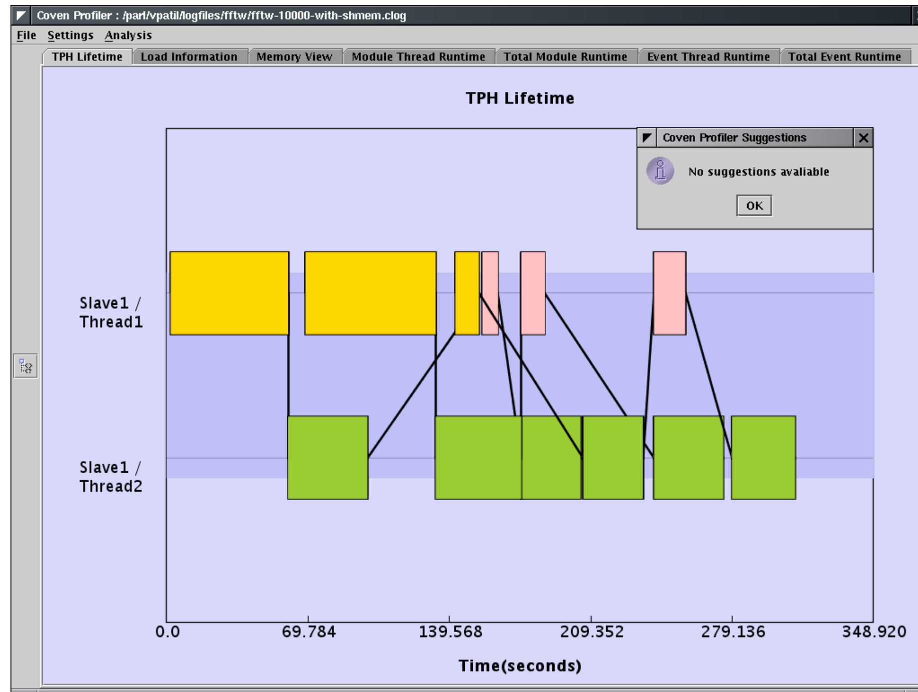

Figure 4.16: TPH view for FFTW application with shared memory

switched off. For simplicity of the visualization we show the runtime information pertaining to a single node (slave1). As seen from the figure the inter thread message transfers took a long time (average of 47.5 seconds, > 5% of the parallel program runtime). Note that in this case the inter thread communication time is overlapped with the module execution time on the destination thread. Also see from the load view for the application in figure 4.15 the Coven modules exhibited a low load while execution indicating that they were spending time waiting for messages.

The expert system was then consulted for any recommendations to improve the performance of the parallel program and it suggested switching on the shared memory feature. The application was then rerun with the shared memory feature on and figure 4.16 shows the corresponding TPH view for it. As seen from the figure the inter thread message time decreased significantly as result of which the overall runtime decreased from 348.9 sec to 310.37 sec. It is important to note that a part of the module execution included the time required to transfer messages among the program

threads. This inter thread message transfer time significantly decreased once the shared memory feature was switched on and hence a performance gain was obtained.

Chapter 5

Conclusions and Future Work

Identification and tuning of performance problems in parallel applications can be difficult, even for parallel computing experts. Since problem solving environments generally target application domain scientists and engineers who are often unfamiliar with the intricacies of parallel application performance tuning, tools which assist them in these tasks are invaluable. In this thesis, we have introduced the Coven's profiling architecture that helps a programmer to tune parallel applications built using Coven. With the help of different examples we have shown how the visualizations provided by the front end tool assist the programmer in identifying the problems that are encountered while developing parallel applications. In addition to his, in order alleviate the requirements of user knowledge to understand and improve the performance of a parallel application, an expert system has been presented. The expert system identifies the common performance problems that are encountered while developing parallel applications using Coven. Once these performance problems have been identified, the expert system tries to help the parallel programmer by making appropriate suggestions to overcome these problems. Currently the expert system is able to identify problems that can be solved by using multi-threading, load balancing and shared memory features in Coven.

Figure 5.1: Current load view for a module

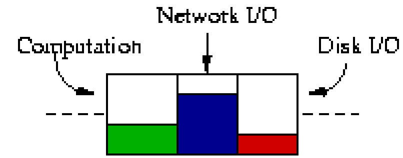

Figure 5.2: Enhanced view for a module with computation, network I/O and disk I/O information

Although the Coven profiling framework is a working system, there exists several directions for it's growth. New visualizations need to be developed that provide detailed performance information such as the exact periods of time for the different types of operations namely computation, disk I/O and network I/O. For instance, currently in case of the load view each rectangular block represents a module and the amount of filled area is proportional to the load exerted by the Coven module during the execution period as shown in figure 5.1. It does not provide any information regarding the amount of time spent doing network and disk I/O activities. It would be useful to divide every block in the load view into three sub blocks with one representing the percentage of the computation time, the second representing the percentage of network I/O time and the third representing the percentage of disk access time as shown in 5.2. These new features can be added by defining additional instrumentation points.

In addition to appending new performance data to the log file, it would be beneficial to develop scalable methods of representing the data. This is especially required since some of the visualizations become cluttered for large number of processors (32 or more). Currently scalability of visualizations is achieved in the Coven profiler by enabling the programmer to filter out unnecessary information. However there are several other ways in which scalability of visualizations can be achieved namely[23]

- Adaptive graphical representation, in which the graphical characteristics of the display are changed in response to the size of the dataset

- Spatial arrangement, in which graphical elements get rearranged so that as a dataset scales, the display size and/or complexity increase at a much slower rate

- Generalized scrolling, in which a localized view of a much larger mass of information is provided

At present, on identifying performance problems, the expert system simply makes suggestions to the programmer in order improve over them. This requires the user to manually edit the Coven program and re-run the application. It would be useful if the profiling system could automate this step. For instance in case the expert system detects that a single threaded application could benefit by using multi-threading, the expert system could perform the control flow analysis of the Coven program and modify the flow graph by placing the modules exerting low loads on separate threads. Similarly if it is detected that a multi-threaded application could be benefited using shared memory, the expert system could switch on the shared memory feature in the middleware and then re-run the application.

Finally, another future line for the expert system is to increment the performance problem base. This can be done by identifying new patterns in the log files that represent a problem and by appending new rules to the knowledge base of the Coven expert system that help in identifying these patterns.

Bibliography

[1] CLIPS. http://www.ghg.net/clips/CLIPS.html.

[2] Papi. http://icl.cs.utk.edu/papi.

[3] Anthony Chan and William Gropp and Ewing Lusk. Performance Visualization for Parallel Programs. http://www-unix.mcs.anl.gov/perfvis/software/log_format.

[4] Anthony Chan and William Gropp and Ewing Lusk. SLOG-2 Software Development Kit . http://www-unix.mcs.anl.gov/perfvis/download/index.htm#slog2sdk.

[5] Anthony Chan and William Gropp and Ewing Lusk. User's Guide for MPE: Extensions for MPI Programs. http://www-unix.mcs.anl.gov/mpi/mpich/docs/mpeman/mpeman.htm.

[6] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille,Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer 28, 11*, 28:37–56, 1995.

[7] Tomas Margalef Burull. *Automatic Performance Analysis of Parallel Programs*. PhD thesis, Department d'Informatica,Universitat Autonoma de Barcelona, Barcelona, Spain, September 2000.

[8] Computer Science Department, University of Oregon. TAU tools. http://www.cs.uoregon.edu/research/paracomp/tau/tautools.

[9] Craig Smith. Java Expert System Shell. http://herzberg.ca.sandia.gov/jess/.

[10] Nathan Debardeleben. *Coven: A Computational Model and Problem Solving Environment Framework for Supporting Optimization of Parallel Applications*. PhD thesis, Computer Engineering,Clemson University, Clemson, SC, USA, July 2004.

[11] J. Dvorak. Using clips in the domain of knowledge-based massively parallel programming, 1992.

[12] Guy Robinson. Load Balancing as an Optimization Problem . http://www.netlib.org/utk/lsi/pcwLSI/text/node248.html.

[13] J. K. Hollingsworth and B. P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 185–194, July 1993.

[14] Sanjeev Krishnan and Laxmikant V. Kale. Automating parallel runtime optimizations using post-mortem analysis. In *Proceedings of the 10th international conference on Supercomputing*, pages 221–228. ACM Press, 1996.

[15] Kei-Chun Li and Kang Zhang. A knowledge-based performance tuning tool for parallel programs.

[16] MCS Division, Argonne National Laboratory. MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich.

[17] Walter B. Ligon III Nathan A. DeBardeleben, Vishal Patil. Using coven to profile and tune parallel programs. Technical report, Parallel Architecture Research Lab, Clemson, SC 29631, USA, February 2004.

[18] Pallas. Vampir/VampirTrace. http://www.pallas.com/e/products/ vampir/documents.htm.

[19] Michael Quinn. *Parallel Programming in C with MPI and OpenMP*. Tata Mcgraw Hill, 2003.

[20] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.

[21] The Computer Science and Oak Ridge National Laboratory Mathematics Division (CSM). Parallel virtual machine. http://www.cs.uoregon.edu/research/paraducks/papers/shpcc94.d.

[22] Dr Lorna Smith. Comparison of code development tools on clusters. http://www.epcc.ed.ac.uk/overview/publications/ training_material/tech_watch/99_tw/ techwatch-clustertools/tools-1.html.

[23] Steven T. Hackstadt, Allen D. Malony, Bernd Mohr. Scalable Performance Visualization for Data-Parallel Programs. http://www.cs.uoregon.edu/research/paracomp/tau/tautools.

[24] Sun Microsystems. Java Foundation Classes (JFC/Swing). http://java.sun.com/products/jfc/index.jsp.

[25] Michela Taufer. *Inverting Middleware: Performance Analysis of Layered Application Codes in High Performance Distributed Computing*. PhD thesis, SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH, Zurich, Switzerland, 2002.

[26] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.