View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Waterloo's Institutional Repository

Parallel Pattern Search in Large, Partial-Order Data Sets on Multi-core Systems

by

Olufisayo Ekpenyong

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2011

© Olufisayo Ekpenyong 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Monitoring and debugging distributed systems is inherently a difficult problem. Events collected during the execution of distributed systems can enable developers to diagnose and fix faults. Process-time diagrams are normally used to view the relationships between the events and understand the interaction between processes over time. A major difficulty with analyzing these sets of events is that they are usually very large. Therefore, being able to search through the event-data sets can enable users to get to points of interest quickly and find out if patterns in the dataset represent the expected behaviour of the system.

A lot of research work has been done to improve the search algorithm for finding event-patterns in large partial-order datasets. In this thesis, we improve on this work by parallelizing the search algorithm. This is useful as many computers these days have more than one core or processor. Therefore, it makes sense to exploit this available computing power as part of an effort to improve the speed of the algorithm. The search problem itself can be modeled as a Constraint Satisfaction Problem (CSP). We develop a simple and efficient way of generating tasks (to be executed by the cores) that guarantees that no two cores will ever repeat the same work-effort during the search. Our approach is generic and can be applied to any CSP consisting of a large domain space. We also implement an efficient dynamic work-stealing strategy that ensures the cores are kept busy throughout the execution of the parallel algorithm. We evaluate the efficiency and scalability of our algorithm through experiments and show that we can achieve efficiencies of up to 80% on a 24-core machine.

Acknowledgements

I would like to thank my supervisor Prof. David Taylor for his guidance and direction during my research work. His diligence and attention to detail is greatly appreciated. I would also like to thank the readers of my thesis, Dr. Paul Ward and Dr. Martin Karsten for their insightful corrections and help towards perfecting this thesis.

Next, I would like to acknowledge Dr. Tim Brecht for his ever timely assistance in providing me with the computing resources I needed to perform my experiments. Without his help, the evaluation of the work in this thesis would have been limited. Furthermore, I would like to thank the faculty and graduate students of the Shoshin Lab at the University of Waterloo for providing an environment where I could share some of my ideas. Their suggestions were useful during the course of my research work.

I would like to appreciate my family; my husband for constantly supporting me in my academic pursuit, and my parents for being my inspiration. Your continuous interest and helpful advice throughout my academic career is one I can always rely on. Thanks also to my siblings who always helped me with errands so that I could spend more time on my research work. All your help, love and prayers made it easier for me to focus on my research.

Finally, I would like to thank God for inspiring me with the confidence I needed to pursue my research efforts. You are my Rock and One that I would forever be grateful to.

Table of Contents

List of Figures .... **vii**

List of Tables .... **ix**

Chapter 1 Introduction **1**

1.1. Problem Statement 1

1.2. Thesis Contributions 2

1.3. Thesis Overview 3

Chapter 2 Event Models in Distributed Systems **4**

2.1. Debugging Distributed Systems 4

2.2. POET Architecture 5

2.3. Event Precedence 8

2.4. Pattern-Search in POET 16

2.4.1. The Pattern Language 17

2.4.2. Convex Closure versus Re-written Patterns 21

Chapter 3 Related Work **24**

3.1. Introduction 24

3.2. Parallelizing CSPs 26

3.2.1. Static Search Tree Distribution 26

3.2.2. Dynamic Work-stealing 29

Chapter 4 Pattern-Search Parallelization **30**

4.1. Introduction 30

4.2. Naïve Backtracking Algorithm 31

4.2.1. Cost Analysis 32

4.3. Parallel Algorithm 34

4.3.1. Task Generation 35

4.3.2. Rules for Task Generation 37

4.3.3. Task-Generation Implementation 44

4.3.4. Memory Localization 51

4.3.5. Cost Analysis of Task Generation 52

4.4. Optimization for Universal Quantifiers ... 53
4.5. Parallel Pattern-Search Architecture ... 54
Chapter 5 Experiments and Results ... 57
5.1. Test Setup ... 57
5.2. Performance Evaluation ... 60
5.2.1. Evaluation of Task-Generation Strategies and Optimizations ... 63
5.2.2. Parallel-Algorithm Evaluation ... 68
5.2.3. Task-Size Analysis ... 71
5.3. Summary ... 72
Chapter 6 More Improvements and Experiments ... 74
6.1. Dynamic Work-stealing Algorithm ... 74
6.1.1. Task Splitting ... 76
6.2. Performance Evaluation ... 77
6.3. Scalability of the Parallel Algorithm ... 80
Chapter 7 Closing Remarks ... 85
7.1. Conclusions ... 85
7.2. Future Work ... 86
7.2.1. Improvements to Variable Re-ordering ... 86
7.2.2. Improvements to the Backtracking Algorithm ... 87
7.2.3. Improvements Based on Re-written Patterns ... 87
7.2.4. Lower and Upper Bounds of Tasks ... 87
7.2.5. Writing POET Patterns ... 88
Appendix A ... 89
Appendix B ... 91
References ... 92

List of Figures

Figure 2.1: The Architecture of POET 6
Figure 2.2: A Process-Time Diagram 7
Figure 2.3: Fidge/Mattern Timestamps Example 10
Figure 2.4: Abstract Events View 12
Figure 2.5: Non-Convex Compound Event 14
Figure 2.6: Convex Compound Event 15
Figure 2.7: Grammar for the POET Pattern Language 18
Figure 2.8: Pattern Language Example 21
Figure 2.9: Pattern Parse Tree 22
Figure 3.1: Search Tree of a CSP 27
Figure 3.2: Task-Distribution Methods 28
Figure 4.1: Grouped vs. Scattered Task-Generation Methods 36
Figure 4.2: Task-Generation Example 38
Figure 4.3: Task-Generation Example 39
Figure 4.4: Grouped Approach – Enforcing Rule 4.2 45
Figure 4.5: Scattered Approach – Case 2 ($S \leq |D_d|$) 49
Figure 4.6: Scattered Approach – Case 3 ($S > |D_d|$) 50
Figure 5.1: PVM Life Event Data Set 58
Figure 5.2: Command-Line Parameters for POET Tools 59
Figure 5.3: Search-Tool Example 59
Figure 5.4: PVM Patterns 61
Figure 5.5: µC++ Patterns 62
Figure 5.6: Optimized Parallel Algorithm for Universal Quantifiers 65
Figure 5.7: Optimized Parallel Algorithm for Universal Quantifiers 66
Figure 5.8: Optimized Parallel Algorithm for Universal Quantifiers 67
Figure 5.9: Optimized Parallel Algorithm for Universal Quantifiers 68
Figure 5.10: Speed-up Grouped Approach (Sub-task Factor of 2) 70
Figure 5.11: Speed-up Using Various Sizes of Tasks 72

Figure 6.1: Task-splitting 77
Figure 6.2: “FinalDataTransfer” Pattern 78
Figure 6.3: Random Patterns 79
Figure 6.4: SendRecv - Speed-up on Multiple Cores 82
Figure 6.5: FinalDataTransfer - Speed-up on Multiple Cores 83
Figure 6.6: ConSendP1P9 - Speed-up on Multiple Cores 84

List of Tables

Table 2.1: ASCII Format of Pattern Language ... 20
Table 4.1: Grouped Task Generation Example (Independent Work) ... 38
Table 4.2: Scattered Task Generation Example (Duplicate Work) ... 39
Table 4.3: Grouped Task Generation Example (Duplicate Work) ... 39
Table 5.1: Execution Time for Sequential and Parallel Algorithms on 2 Cores ... 63
Table 5.2: Execution Time for Sequential and Parallel Algorithms on 4 Cores ... 63
Table 5.3: Execution Time of Non-Optimized vs. Optimized Algorithms on 2 Cores ... 64
Table 5.4: Execution Time of Non-Optimized vs. Optimized Algorithms on 4 Cores ... 64
Table 5.5: Total time of the Parallel Algorithm ... 71
Table 6.1: Execution Time for Static vs. Dynamic Strategies on 4 Cores ... 80
Table 6.2: Execution Time of Parallel Algorithm on Several Cores ... 81
Table 6.3: Speed-up of Parallel Algorithm on Several Cores ... 81

Chapter 1

Introduction

Applications developed to run with various components on several computers have been around for many years. The benefits of such distributed applications are important to both software engineers and end-users. To software engineers, distributed systems make it possible to design applications in which components are decoupled. This makes software maintenance and re-use more feasible. To the end-users, distributed systems are more scalable and provide a lot of performance benefits. In addition, the prevalent use of multi-core computers has also increased the desire for more applications to work seamlessly across several cores.

1.1. Problem Statement

The benefits of distributed systems are accompanied by challenges. Such systems are more complex than standalone applications and are more challenging to monitor and debug. This complexity arises from the unpredictable behavior of the system, caused by the execution of concurrent programs and the absence of any guarantees about the way their execution will interleave. This makes it more difficult for developers to reproduce a problem, thereby increasing the time and effort required to diagnose and fix a bug.

In order to aid the debugging of distributed systems, they are usually set up to emit logging information that describes their execution. These logs are usually very large containing a copious amount of events. This makes it more difficult to analyze and

understand the execution history of the system when trying to fix faults. Another difficulty with analyzing such logs is that distributed systems have no global clock. Therefore, when comparing the timestamp of an event occurrence on one computer to that of another event that occurred on another computer, one may come to erroneous conclusions because these timestamps may not necessarily reflect the order in which the events actually occurred. To this end, several tools have been developed to make debugging of these systems easier. This thesis focuses on improving one such tool.

1.2. Thesis Contributions

In this thesis, we improve a monitoring and debugging tool called the Partial-Order Event Tracer (POET). POET is capable of representing the execution history of distributed systems in a partial-order using logical timestamps. It provides facilities for both offline and online monitoring of the system, viewing process abstractions or clusters, and viewing abstract or compound events. In addition, POET provides an expressive language that enables users to specify complex patterns and search for them in large event datasets. This can be useful when diagnosing faults such as performance bottlenecks, race conditions or improper access of resources arising due to poor synchronization among threads. Extensive research work has gone into the development of this tool [8, 13, 22, 25, 32, 35, 36, 39, 42] but more improvements are needed.

Our work focuses on improving the search algorithm for finding event patterns of interest by parallelizing the algorithm for execution on a shared-memory multi-core system. As we will see later, the search problem in POET can be modeled as a Constraint Satisfaction Problem (CSP) and we use this model as a basis for developing an efficient parallel algorithm. The main thesis contributions are as follows:

a) We employ the technique introduced by Habbas et al. [20] for distributing the search space (i.e., the large event dataset) into several tasks that can be handled by various cores. We extend Habbas' technique by developing a method for task distribution that can be applied to CSPs with a large domain space. We include

rules to be used during task generation in order to avoid duplicate work-effort among the cores.

b) We analyze two approaches for search-space distribution and make a recommendation as to which approach is more suitable for the pattern-search problem in POET.
c) We provide optimizations to the parallel algorithm in order to deal with patterns that have certain unique properties.
d) In order to ensure that all processors are busy throughout most of the algorithm's execution time, we develop a hybrid method of task distribution by initially dividing the search space into tasks before processors begin the search and then allowing processors to steal work from others as they become idle, otherwise called dynamic work-stealing.
e) Finally, we show in experimental results that the parallel algorithm is scalable providing efficiencies of up to 80% on 24 cores.

1.3. Thesis Overview

This thesis is organized as follows: Chapter 2 provides an overview of POET, detailing its architecture and its major features. It includes the algorithm used for generating logical timestamps, the language used for representing event patterns, and the algorithms used in the search. Chapter 3 introduces related work in the area of parallelizing CSPs. Chapter 4 describes in detail the approaches we developed for generating tasks, the rules involved, and the optimizations to the parallel algorithm. Chapter 5 shows the experimental results obtained from the parallel algorithm and makes recommendations on the number of tasks to generate given a certain number of cores. Chapter 6 describes the dynamic work-stealing algorithm used for load-balancing. It also includes some experimental results of the work-stealing algorithm as well as tests showing the scalability of the parallel algorithm. We conclude with Chapter 7, which provides a summary of the results and identifies areas of future work.

Chapter 2

Event Models in Distributed Systems

2.1. Debugging Distributed Systems

As previously mentioned, debugging distributed systems is a very hard problem and several techniques have been explored in order to tackle this problem. Offline approaches describe the situation where logs from the distributed system are analyzed after the system's execution has terminated. A common approach is for the various nodes in the system to send log information to a central server. The logs can then be retrieved from the server and analyzed for violations of specific properties. Certain tools like Pip [34] make it possible to specify the requirements of the system beforehand using a declarative language and then the logs can be checked for violations of these requirements. Although there is only a small performance overhead due to local logging with this kind of approach, it may be generally difficult for a programmer to specify the requirements of the system using the language.

Another offline approach generally used is the replay technique [16, 28, 41, 42]. In this case, trace information is collected from the various nodes and the execution of the system can be replayed in order to reproduce non-deterministic errors such as race conditions. There is generally a lot more overhead in logging, as more information is necessary in order to replay the execution path of the deployed system. It is also difficult to replay certain events such as shared-memory access or thread scheduling. Another approach to distributed debugging involves the use of virtual machines [21, 28]. In this approach, there is a virtual machine located above the hardware, and between the

application and the operating system. The nodes in the distributed system and any network latencies are then simulated on the virtual machine. This approach provides flexibility in debugging (for example, nodes in the system may be slowed down or stopped when trying to reproduce a race condition), but it may be difficult to efficiently simulate all the nodes in a large distributed system on one machine.

Online approaches to debugging usually involve first specifying properties of the system that should not be violated and then checking the system for violations of these properties during execution. This method usually involves taking a consistent snapshot of the execution and checking for violations in the state recorded in a snapshot. One challenge involved here is understanding what size of snapshot is adequate, i.e., a global snapshot across all nodes in the system or just of neighbouring nodes [27, 40]. Also, a lot of care must be taken in order not to introduce too much performance overhead that will significantly alter the normal execution of the distributed system.

In this thesis we focus on the offline approach to distributed debugging by making use of the Partial-Order Event Tracer (POET). POET is a tool that was developed to collect and analyze large traces from distributed systems. POET supports both the offline and online approaches to monitoring and debugging distributed systems. It was originally developed in 1991 by the Shoshin Research Group, and the original implementation was written in C and C++. An alternate implementation is written in Java as a plug-in to Eclipse [1]. This thesis uses the Java version, called Eclipse POET. In the next section, we discuss the architecture of POET in detail.

2.2. POET Architecture

POET itself is a distributed system consisting of several processes. The target programs being instrumented submit events to the event server. These events provide a good view of the execution history of the programs. The event server interacts with a viewer and a checkpoint process (described below). The viewer communicates with the event server to retrieve POET events and then timestamps the events in order to display them on a process-time diagram.

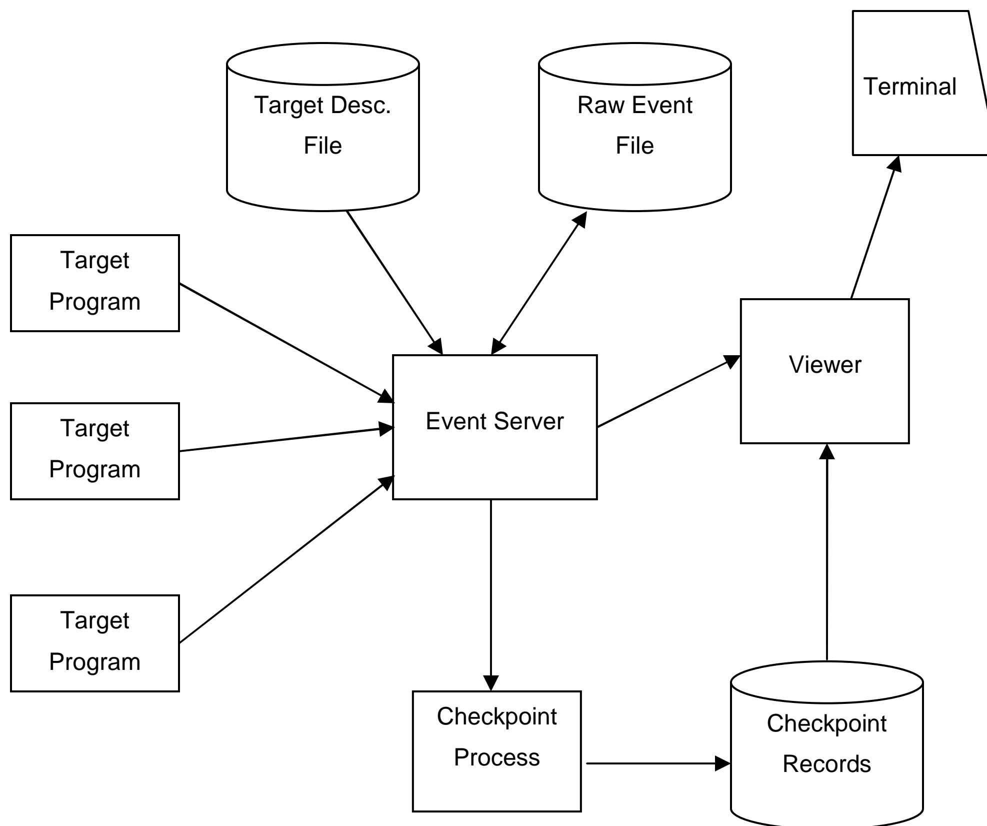


Figure 2.1: The Architecture of POET

In distributed systems, a process-time diagram is used to visually represent the interaction between processes that occurs through message passing via "sends", "receives" or some other sort of messages. Figure 2.2 shows an example of such a diagram. Each horizontal line referred to as a "trace" represents a process's execution in time advancing from left to right. A unary event as shown in the diagram is one that is not involved in any interaction with other processes. A synchronous event is viewed as a single event that occurs simultaneously on two processes with the message exchange occurring at the same time on both processes. Vertical directed lines between two single events are used to represent synchronous communication. Asynchronous events are viewed as originating from one process and terminating on another, and they are used to model asynchronous communication between two processes. Slanted directed lines represent asynchronous communication.

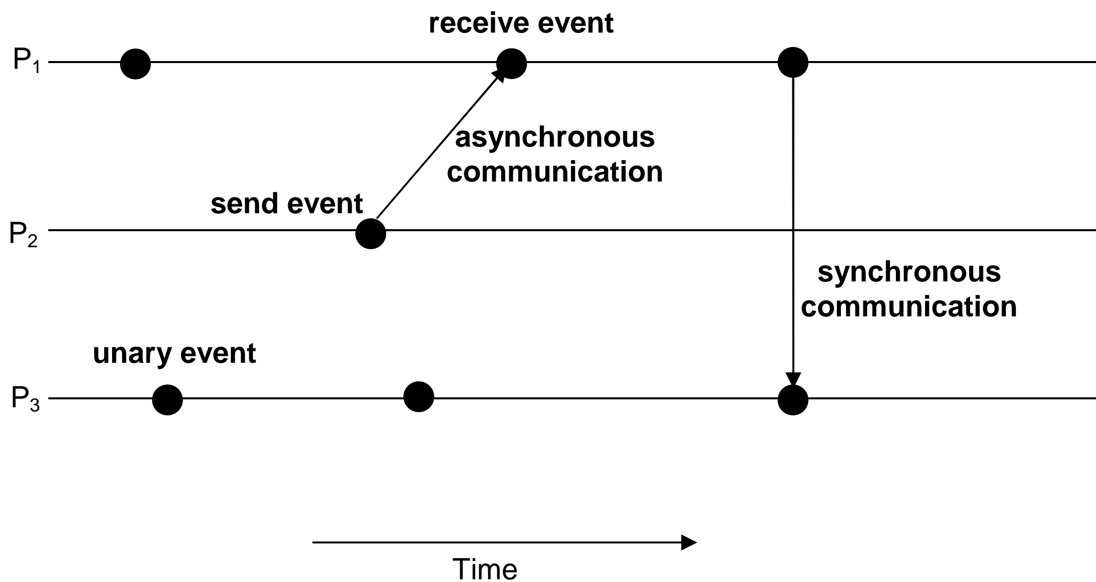


Figure 2.2: A Process-Time Diagram

In order for the viewer to draw the process-time diagrams, the events must first be timestamped. Timestamping is done using vector timestamps (as introduced by Fidge [15] and Mattern [29]) which grow in size as the number of processes increases. Though more time-consuming, timestamping the events is done by the viewer because performing it at the server would require a lot of disk space to store the timestamps. Due to the runtime costs of timestamping at the viewer, the checkpointer process receives each event from the server and timestamps it. It then periodically writes out a snapshot of the internal state of the timestamping algorithm. These snapshots are subsequently used by the viewer to speed up the process of timestamping the events when drawing the display.

POET was developed to be target-independent and this is achieved through the use of target-descriptor files. These files are used to describe the native events emitted by the target programs and map them to POET events. The event server converts the events into binary raw-event format in EF files. These files can be converted into a more portable format stored in ASCII text in a UEF file. As we will see later, these UEF files can be imported into the viewer in Eclipse POET and displayed on the screen.

2.3. Event Precedence

POET uses Fidge/Mattern vector timestamps in order to efficiently determine the precedence relationships or causality between primitive events. Developed by Fidge [15] and Mattern [29], vector timestamps make it possible to determine precedence relationships between events in constant time. In this thesis, we focus on two precedence relationships: the happened-before and concurrent relationships as introduced by Lamport [26].

Definition 2.1:

The happened-before relation is denoted by $\rightarrow$, and a primitive event is said to happen before another primitive event if any of the following holds:

1. If a and b are events on the same process and a occurs before b, then $a \rightarrow b$.
2. If a is the send event representing the sending of a message by one process and b is the receive event representing the receipt of that message on another process, then $a \rightarrow b$.
3. The happened-before relation is transitive, i.e., if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Definition 2.2:

The concurrent relation is denoted by $\|$, and a primitive event, a, is concurrent with another primitive event, b, if a does not happen before b and b does not happen before a (i.e., $a \mathrel{!\!\rightarrow} b \wedge b \mathrel{!\!\rightarrow} a$).

Definition 2.3:

Where a pair of events indicates a synchronous or asynchronous communication between two traces, we say that one event is the *partner* of the other. Partner events are denoted by $a.b$.

The above definitions are sufficient to define the relationship between primitive events. It is however useful to note that when determining precedence relationships, we are mostly interested in comparing primitive events that are not equal. A primitive event is uniquely identified by a trace number and an event number, so an event denoted as a_1^2, implies that this is the second event that occurred on the first trace (i.e., the subscript is the trace number and the superscript is the event number). To determine precedence relationships, associated with each primitive event is a timestamp vector, *V*, of *n* integers, where *n* is the number of traces in the distributed system being monitored. Each process, P_i, maintains a local clock vector denoted by C_i of size *n* which is used for timestamping primitive events. In POET, primitive events are timestamped following the algorithm proposed by Fidge as shown below:

1. Each clock vector, C_i is initialized to zero at the beginning of the computation for each process P_i.
2. Whenever process P_i performs a unary event *a*, its local clock is incremented by 1 and the timestamp of the event V_a is equivalent to C_i, i.e.,

   $C_i [i] = C_i [i] + 1$

   $V_a = C_i$

3. When a process P_i sends an asynchronous message represented by the event *a*, it updates its local clock and timestamp as for unary events and attaches the timestamp to the message.
4. When a process P_j receives the asynchronous message with the timestamp (now denoted as $C_i^{'}$), it increments position *i* of $C_i^{'}$ and position *j* of its local clock, C_j, by 1 and update the entries in C_j to the maximum of its current value and the new values in $C_i^{'}$. If *b* is the receive event then its timestamp V_b, will be set to the updated value of C_j. Formally

   $C_i^{'}[i] = C_i^{'}[i] + 1$

   $C_j [j] = C_j [j] + 1$

   $\forall p \in \{1,...,n\}, C_j[p] = \max(C_j [p], C_i^{'}[p])$

   $V_b = C_j$

5. If a is a synchronous send event of P_i and b is the corresponding receive event on P_j, then the local clocks C_i and C_j are set to the maximum of each of the entries in C_i and C_j. This can be achieved by the receiver sending a confirmation message with its local vector clock back to the sender so that the sender can update its local clock.

   $C_i [i] = C_i [i] + 1$, sender updates local clock upon sending message

   $C_j [j] = C_j [j] + 1$, receiver updates local clock upon receipt

   $\forall p \in \{1,...,n\}, V_a [p] = V_b [p] = \max(C_i [p] , C_j[p])$

As an extension to Fidge's algorithm, Cheung [13] proposed that in preparation for the next event, the processes involved in the synchronous communication should increment the element in its local clock of the partner process, i.e.,

$C_i [j] = C_i [j] + 1$

$C_j [i] = C_j [i] + 1$

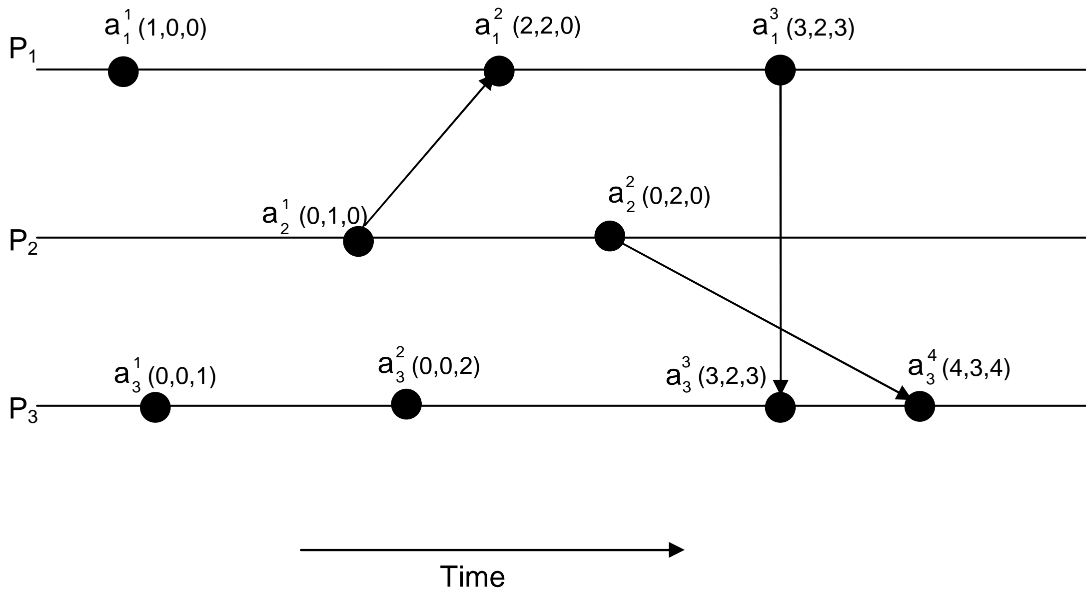


Figure 2.3: Fidge/Mattern Timestamps Example

Figure 2.3 shows an example of a process-time diagram with the vector timestamps following the algorithm above. Here we see that a_1^1 is an example of a unary event and following Step 2 of the algorithm, its timestamp will be (1,0,0). According to Step 4, when P_1 receives the asynchronous message from P_2 it will increment its local clock to (2,0,0) and the timestamp from P_2 to (0,2,0). Taking the maximum of both clocks will yield a timestamp of (2,2,0) for the receive event a_1^2. For the synchronous message from P_1 to P_3, Step 5 will be applied. The local clock at P_1 upon sending the message is (3,2,0). Upon receipt of the event at P_3, its clock will be set to (0,0,3). Taking the maximum of each entry of the two clocks at P_3 (i.e., maximum of (3,2,0) and (0,0,3)) will yield a timestamp of (3,2,3) for the receive event, a_3^3. P_3 will send a confirmation message to P_1 along with its local clock enabling P_1 to update the timestamp for a_1^3 to (3,2,3). Both P_1 and P_3 will increment the other's entry in their local clocks according to the rule by Cheung. Hence, the receive event a_3^4 at P_3 is (4,3,4) and not (3,3,4).

The vector timestamps enable us to determine the precedence relationships between primitive events in constant time. To determine if an event, a, occurring on process P_i happens before another event, b, occurring on process P_j, we simply need to check if $V_a[P_i] < V_b[P_i]$. If this is the case then a happens before b. If this is not the case and $V_b[P_j] < V_a[P_j]$, then b happens before a. To test for concurrency between a and b, then we simply need to check that the tests $V_a[P_i] < V_b[P_i]$ and $V_b[P_j] < V_a[P_j]$ are both false. To check if two primitive events, a and b are equal we would need to compare their trace and event numbers. In the example above, we can determine that a_3^2 happens before a_1^3 because $V_{a_3^2}[P_3] = V_{a_3^2}[3] = 2 < 3 = V_{a_1^3}[1] = V_{a_1^3}[P_1]$. Similarly, we compute that a_3^2 and a_1^1 are concurrent because $2 \nless 0$ and $1 \nless 0$.

Process-time diagrams are helpful in enabling the developer to visually inspect the interaction between processes and therefore provide useful information when monitoring and debugging distributed systems. The size of the event data-set however makes it impossible to fit the entire execution on the screen and scrolling through it is also cumbersome. Several approaches have been investigated in order to deal with this problem. One approach is through event abstraction and the other is by searching for event patterns in the data-set (otherwise known as pattern-search).

In event abstraction, primitive events are grouped together into a meaningful unit called an abstract or compound event and displayed graphically. Kunz [25] developed an approach to automatically combine a number of primitive events, possibly from various processes, into a single abstract event. Other manual or semi-automatic approaches that allow the user to specify which events to abstract away were investigated by Seeleman [35] and Seuren [36]. Graphically, an abstract event in POET is represented by a vertical rectangle that stretches over all the processes involved in the event. The intersection between the rectangle and a process that is part of this abstract event will be filled, otherwise an open intersection signifies that no event from the process belongs to this abstract event. For example, in Figure 2.4 primitive events from processes P_1, P_2, and P_5 make up the first abstract event.

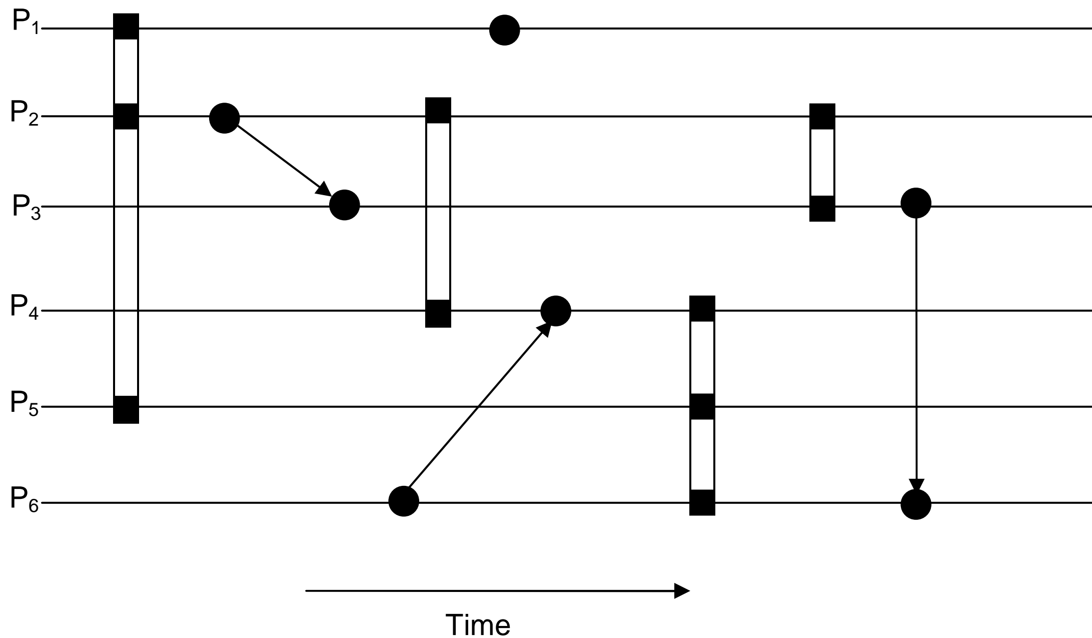


Figure 2.4: Abstract Events View

Certain complications arise when defining precedence relationships between abstract or compound events. In event modeling for distributed systems the following have been proposed for defining the precedence relationships between compound events [22, 25].

Definition 2.4:

A compound event, A, happens before another compound event, B, if all the primitive events in A happen before all the primitive events in B. Formally

$$A \rightarrow B \Leftrightarrow \forall a \in A, \forall b \in B, a \rightarrow b.$$

Definition 2.5:

A compound event, A, happens before another compound event, B, if there exists a primitive event in A that happens before another primitive event in B. Formally

$$A \rightarrow B \Leftrightarrow \exists a \in A, \exists b \in B, a \rightarrow b.$$

Definition 2.6:

A compound event, A, is concurrent with another compound event, B, if neither event precedes the other, i.e.,

$$A \parallel B \Leftrightarrow A \;!\!\rightarrow B \wedge B \;!\!\rightarrow A$$

The first definition of the happened-before relationship (Definition 2.4) was found to be easier to deal with as it maintains the partial order relationship between both primitive and compound events. It was however found to be too restrictive as it was sometimes impossible to define happened-before relationships between compound events that were clearly related. The second happened-before definition (Definition 2.5) was found to be more intuitive though it requires more work to deal with as it breaks the transitivity property, i.e., if $A \rightarrow B$ and $B \rightarrow C$, it does not necessarily mean that $A \rightarrow C$. The definition also contradicts the partial-order relationship as it is possible for a compound event A to happen before B and for B to happen before A. In order to deal with these problems, Kunz [25] introduced the idea of compound events that are *convex*.

Definition 2.7:

A compound event made up of a set of primitive events E is said to be convex if and only if $\forall x, y \in E,\ x \rightarrow z \rightarrow y \Rightarrow z \in E$.

Definition 2.8:

Event sets A and B are disjoint $\Leftrightarrow A \cap B = \varnothing$

Definition 2.7 implies that in a convex event set there are no intervening events that are not included in the set. In Figure 2.5, the set of open-circled events is a non-convex compound event as the dark-filled event is an intervening event. By including the dark-filled event, the compound event set becomes convex (See Figure 2.6).

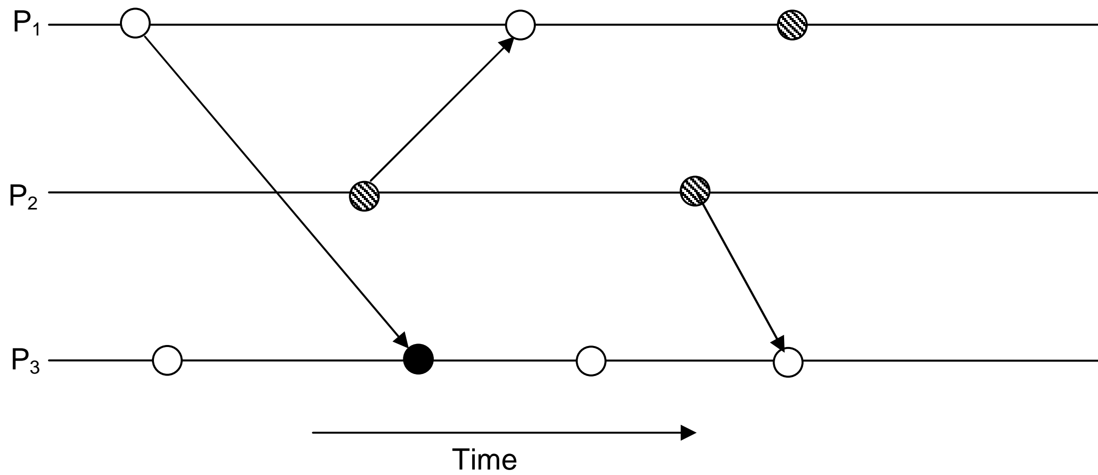


Figure 2.5: Non-Convex Compound Event

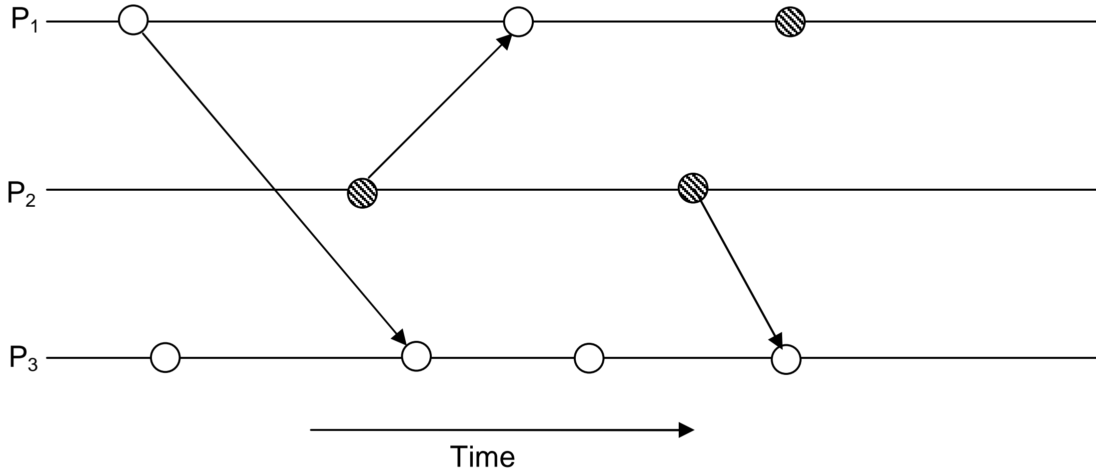


Figure 2.6: Convex Compound Event

By restricting compound events to the set of events that are convex and disjoint (see Definition 2.8), it is possible to maintain the partial-order relationship between compound events (except for transitivity) for event sets that do not *cross* each other (see Definition 2.9 [32]). In other words, given two convex compound event sets *A* and *B*, it is still possible for $A \rightarrow B$ and $B \rightarrow A$ to both be true if *A* and *B* cross each other.

Definition 2.9:

The event set *A* crosses another event set $B \Leftrightarrow \exists x_0 \in A, \exists y_0 \in B \wedge x_0 \rightarrow y_0 \wedge \exists x_1 \in A, \exists y_1 \in B \wedge y_1 \rightarrow x_1 \wedge A$ is disjoint from *B*.

Definition 2.10:

The event set *A* overlaps the event set $B \Leftrightarrow A \cap B \neq \emptyset$

Following Definitions 2.9 and 2.10, two event sets are said to be *entangled* if they either cross or overlap each other. The entanglement operator $\leftrightarrow$ is used to specify that event sets are entangled [32].

Definition 2.11:

$A \leftrightarrow B \Leftrightarrow A$ crosses $B \vee A$ overlaps B

Definition 2.12:

$A \not\leftrightarrow B \Leftrightarrow A$ does not cross $B \wedge A$ is disjoint from B

By ensuring that event sets are not entangled, the following modification to the happens-before relationship between compound events have been proposed [32].

Definition 2.13:

$A \rightarrow B \Leftrightarrow \exists a \in A, \exists b \in B \wedge a \rightarrow b \wedge A \not\leftrightarrow B$

The introduction of compound events is not only useful when abstracting away primitive events to aid visualization of the process-time diagrams, but also when searching for event patterns. Pattern-search is another approach used to cope with the large set of events emitted by distributed systems. Pattern-search allows the user to specify event patterns that are of interest using a pattern language and then a search algorithm finds these patterns in the data-set and displays them to the user. Pattern-search essentially allows the user to jump to points of interest on the screen. The next sections discuss in detail the pattern language used for defining a pattern and the algorithms used to find matches to the pattern.

2.4. Pattern-Search in POET

Before being able to search for events, there must be a well defined way of specifying what is to be found. This is achieved using a pattern language. There has been extensive research work focusing on defining pattern languages that can be used to search for events from distributed systems [22, 32]. The common parts of the proposed languages are the need for a way to specify attributes of an event, compound events or event classes,

precedence and concurrency relationships, and a way to combine various pattern components. The following sub-section describes the pattern language used in POET.

2.4.1. The Pattern Language

In the POET pattern language, the most basic element of the pattern is the event class. An event class is represented by a 3-tuple that describes the process in which the event occurs, the type of the event (e.g., send or receive) and an additional text for including useful information. The tuple is represented as ["<process>", "<type>", "<text>"]. For example, a tuple with all entries empty, ["", "", ""], would capture all the primitive events in the data-set while a tuple such as ["P1", "", ""] represents all the events occurring on process "P1". An event class that has a partner event class would be represented by two 3-tuples separated by a period.

The operators in the language are the happened-before or precedence (→), concurrent (||), and entanglement (↔) operators and they specify the constraints between the event classes forming a *clause*. Another element in the language is the logical operators OR (∨) and AND (∧). These are used to combine the clauses to specify more restrictions on the pattern. For example, a simple pattern such as $(A \rightarrow B) \parallel C \wedge (B \rightarrow C)$ implies that the search algorithm would find the events from event class A that happen before events from B and are concurrent with events from C. Moving on to the next clause, the search algorithm would then find events from B that happen before those from C. Note that the set of events from B and C that satisfy the second clause do not necessarily have to be the same set of events from B and C that satisfy the first clause. Most of the initial building blocks of the pattern language were introduced by Jaekl [22].

predicates	⇒	(predicate “;”)*
predicate	⇒	id “:=” clause
	|	id variable (“,” variable)*
clause	⇒	term basicOperator term
	⇒	term “!” basicOperator term
	|	term booleanOperator term
basicOperator	⇒	“→”
	|	“||”
	|	“↔”
booleanOperator	⇒	“∧”
	|	“∨”
term	⇒	id
	|	variable
	|	class
	|	class.class
	|	“(” clause “)”
class	⇒	“[”process “,” type “,” text “]”
variable	⇒	[“$”, “*”, “~”]id
id	⇒	alpha(alnum)*
alpha	⇒	[“a” – “z”, “A” – “Z”, “_”]
alnum	⇒	[“a” – “z”, “A” – “Z”, “_”, “0” – “9”]
string	⇒	[“a” – “z”, “A” – “Z”, “_”, “0” – “9”, “:”, “ ”, “\t”, “*”, “.”, “””, “(”, “)”]+

Figure 2.7: Grammar for the POET Pattern Language

Nichols [32] introduced the use of variables to the pattern language, which affect how the pattern-search algorithm works. A dollar-sign, $, is used to specify a variable belonging to an event class and allows the search algorithm to bind the primitive events from that class to the variable. This enables the primitive events to be used as the search progresses. To illustrate further, in the previous example, we could replace the event class B with a variable $b resulting in the pattern ($A \rightarrow$ $b) || $C \wedge$ ($b $\rightarrow C$). For this pattern, the search algorithm will behave differently. Here, the primitive events belonging to B that satisfied the constraints in the first clause must be used in satisfying the second constraint. This is more intuitive for the user and is probably more desirable.

Variables could be marked with a universal quantifier (*). This implies that the chosen primitive event in the given clause must satisfy all the primitive events associated with the variable marked with the universal quantifier. For example, given a pattern $a → *b, with $a and *b taken from event classes A and B respectively, implies that the event assigned to $a should precede all the events from event class B. Variables could also be marked with a tilde, ~, which indicates that the events associated with such variables should not be returned as part of the match. So for example, the pattern ~a → B, associated with event classes A and B respectively, implies that the search algorithm should find events from A that happen before events from B but only return the events from B to the user or the next level of the pattern matching process.

The limited operator which was created much earlier by Jaekl is no longer used because the introduction of variables and universal quantifiers is sufficient to replace the limited operator. The limited operator given in an example by $A \overset{C}{\rightarrow} B$, means that the search algorithm should return only events in A that precede events in B, where no occurrence of an event matching C happens both after the match to A and before the match to B. With universal quantifiers, this pattern can now be written as ($a → $b) ∧ ($a !→ *c ∨ *c !→ $b). Figure 2.7 shows the grammar for the current pattern language used in POET.

In POET, patterns are specified in an ASCII plain-text file called the pattern file. The following table shows a mapping between the formal notation of the language and its ASCII format.

Table 2.1: ASCII Format of Pattern Language

	Formal	**ASCII**
Happens-before	$\rightarrow$	-->
Concurrent	||	||
AND	$\wedge$	&
Limited	$A \xrightarrow{C} B$	A - (C) -> B

Next, we discuss certain features provided by the pattern language that simplify writing the pattern file and make it easier to read and follow. Consider a distributed application consisting of a server and two clients communicating using TCP sockets. The clients establish a connection with the server and then begin sending messages to the server. Each client sends 20 consecutive messages to the server (to fill up a buffer) and then waits until there is space in the buffer before sending more messages. The clients can also receive messages from the server. The clients close the connection after a certain number of messages have been sent. A programmer monitoring or debugging this application can verify the connections made to the server using the patterns defined in Figure 2.8.

In this figure, it is seen that in writing patterns, one can "declare" variables in much the same way as is done in most programming languages. Such declarations make it easier to refer to patterns in another more complex pattern. For example, we see that "StartConnect" and "DoneConnect" are event classes represented by a 3-tuple as described earlier. "StartConnect" given by ["", "Accept", ""] means that this event class would match any events occurring on any process with an event type of "Accept" and with any associated description. The following tuple after the period describes the partner event type "Accept_stream" that is associated with each "Accept" event. Line 3 in the figure declares variables `$sc` and `*sc_all` as associated with the "StartConnect" event

class. “ConnectionEstablished” is a pattern that is made up of $sc, $dc and *sc_all variables. Much like “StartConnect”, after “ConnectionEstablished” is defined, it can represent a “type” for declaring other variables as is seen in Line 6 where *ce_all and $ce are declared. The *ce_all and $ce variables can then be used in another pattern as in “FirstConnectionEstablished”. The patterns in Lines 5, 7 and 8 enable one to determine how many clients established connections to the server, the first connection established and the last connection established, respectively. This is just an example of what the pattern language allows users to specify and it also shows that the language is flexible enough to allow writing very complex patterns that contain compound events.

```
1.    StartConnect := ["", "Accept", ""].["", "Accept_stream",""];
2.    DoneConnect := ["", "Accept_done", ""].["", "Accept_done_stream",""];
3.    StartConnect $sc, *sc_all;
4.    DoneConnect $dc;
5.    ConnectionEstablished := ($sc --> $dc) & ((*sc_all !--> $dc) | ($sc !--> *sc_all));
6.    ConnectionEstablished  *ce_all, $ce;
7.    FirstConnectionEstablished := (*ce_all !--> $ce);
8.    LastConnectionEstablished := $ce !--> *ce_all;
```

Figure 2.8: Pattern Language Example

2.4.2. Convex Closure versus Re-written Patterns

We discussed earlier the need to introduce convex events in order to maintain the partial-order relationships between compound events. In this section, we will see the implication of searching for events that make use of patterns involving compound events. We consider two approaches to pattern-search: one that takes the convex closure of the compound events during pattern matching and another approach that reduces complex patterns into a simpler format.

Earlier work in POET about searching for patterns that contain compound events involved finding the convex closure of the matching set of events during the search algorithm [8, 39]. Recall that a set of events is said to be convex if there is no intervening event not included in the set that happens before an event in the set and after another event in the set. If such an intervening event exists then it must be included in the set of events to make it convex. It is clear from the previous section that working with convex event sets that do not cross each other enables us to make meaningful precedence tests between compound events and avoid situations where a compound event happens both before and after another compound event. We now move on to examine searching for event patterns that contain compound events.

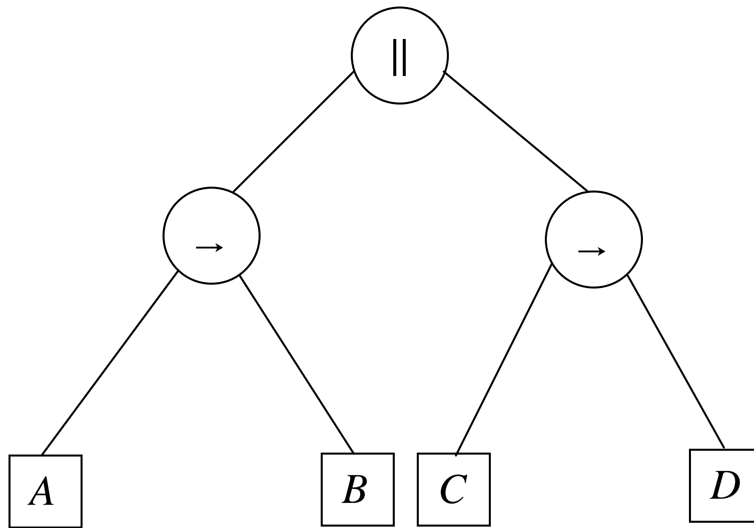


Figure 2.9: Pattern Parse Tree

Given the pattern $(A \rightarrow B) \parallel (C \rightarrow D)$ as represented by the pattern parse tree shown in Figure 2.9, the search algorithm begins by assigning primitive events associated with event classes A and B at the leaves of the tree. These events are then filtered keeping only those that satisfy $A \rightarrow B$. The algorithm would then find the convex closure of these remaining pairs of events. Similarly, the algorithm would find events that match $C \rightarrow D$ and then find their convex closure. With the convex event sets from the left-hand and

right-hand side of the tree, the algorithm would then determine which events satisfy the concurrent operator according to Definition 2.6 and return this set to the user. Experiments showed that finding the convex closure is the most expensive part of the search algorithm [8, 32].

Nichols [32] showed that it is possible to eliminate the need for the finding the convex closure by rewriting the pattern (based on certain rules) into a simpler format that consists of at most a 2-level hierarchy. The rewritten pattern would consist of a set of happens-before relations in conjunctive normal form (CNF). Rewriting is done using Definition 2.5 of compound events. For example, given the pattern $(A \rightarrow B) \parallel (C \rightarrow D)$[1], we begin from left to right and rewrite the pattern as follows. The first component, $A \rightarrow B$, will be assigned variables and re-written as `$a → $b`. The same will be done for the next component to get `$c → $d`. Then by definition of concurrency between compound events (Definition 2.6), we can expand the pattern and add the following restrictions: `$a !→ $c`, `$a !→ $d`, `$c !→ $a`, `$d !→ $a`, `$b !→ $c`, `$b !→ $d`, `$d !→ $b`, `$c !→ $b`, `$a ≠ $c`, `$a ≠ $d`, `$b ≠ $c`, and `$b ≠ $d`. The conjunction of all these constraints will form the re-written pattern.

Furthermore, patterns containing variable modifiers or logical operators like *, ~, !, ∨, and ∧ can be rewritten by applying the definition for compound events, De Morgan's laws, and the mathematical methods for converting boolean expressions into CNF form. Though re-written patterns tend to be very verbose, experiments showed that pattern-search using this approach is substantially faster than finding the convex closure [32]. For this reason, we only consider re-written patterns in this thesis.

Having gone through some background information that describes the application used in this thesis, we move on to previous research work that is more closely related to the focus of this thesis. We formally define the pattern-search problem and provide a good theoretical background that will enable us to better understand how to parallelize the pattern-search algorithm.

---

[1] Example taken from page 101 of Matthew Nichols' thesis [32].

Chapter 3

Related Work

Several problems such as the pattern-search problem in POET can be formalized as Constraint Satisfaction Problems (CSPs). In this section, we define CSPs, and then briefly describe current sequential algorithms used for solving these problems. Finally, we discuss how to parallelize the algorithms for solving this class of problems.

3.1. Introduction

A CSP is defined by a set of variables *X*, a set of constraints *C*, and a set of domains *D*. Each domain is associated with a variable and contains the allowable values for the variable. Solving a CSP involves finding an assignment of values to the variables in order to satisfy the given set of constraints. More formally, a CSP is given by the following definition [31].

Definition 3.1:

A Constraint Satisfaction Problem ***P*** is given as a tuple $\boldsymbol{P = (X, D, C, R)}$ where

- $X = \{x_1, x_2, ..., x_n\}$ is a set of *n* variables.
- $D = \{D_1, D_2, \ldots, D_n\}$ is a set of *n* domains and each D_i is associated with X_i.
- $C = \{C_1, C_2, \ldots, C_m\}$ is a set of *m* constraints where each constraint C_i is defined by a set of variables $\{x_{i1}, x_{i2}, ..., x_{in_i}\} \subseteq X$.

- $R = \{R_1, R_2, ..., R_m\}$ is a set of m relations where each relation R_i defines a set of n_i-*tuples* on D_{i1} x D_{i2} x ... x D_{in_i} compatible with respect to C_i. In other words, a relation R_i defines the combination of values for each of the variables that satisfy a constraint C_i.

CSPs are NP-complete [12] because they require an exhaustive search to find a solution and the most basic approach is to use a naïve backtracking algorithm. In this algorithm, the first step is to find a valid value to assign to the current variable. Once a value is found, the algorithm picks the next variable and finds a valid value to assign to it that does not conflict with the previously assigned variable(s). If a valid value cannot be found, the algorithm backtracks to the last variable and assigns another value hoping that this new value will lead to a successful assignment of the next variable. This process is repeated until all variables have been assigned. The major limitation of the basic backtracking algorithm is that it is exponential in the number of variables, therefore several optimizations have been proposed. For example, a heuristic that uses a static reordering of variables so that a "good" variable is chosen as the first variable for assignment has the effect of reducing the runtime of the search algorithm [38].

Another more intelligent approach to solving CSPs is called back-jumping [17] This approach is similar to the naïve backtracking algorithm except that during the backtrack step, the algorithm jumps to the variable that is hindering the algorithm from moving forward. This results in cost savings as the algorithm quickly picks the next value of the variable that is the point of failure as opposed to simply choosing the next value of the last variable that was assigned. The naïve algorithm that chooses the last variable that was assigned may slow down the progress of the algorithm towards finding a solution.

Another approach called dynamic backtracking [17] is a variation of back-jumping but instead of losing all the work done after the point of failure, dynamic backtracking preserves this work. In other words, when the algorithm "jumps" over variables that have already been assigned, and then re-assigns a new value to the variable that is the point of failure, it does not change the values of the variables that have already been assigned if they are not in conflict with the new value. Several other optimizations and techniques that exist for solving CSPs [6, 7, 23] are beyond the scope of this thesis.

3.2. Parallelizing CSPs

In addition to optimizing the sequential algorithms, additional research work has focused on parallelizing such algorithms as a way to improve performance [10, 11, 24, 30, 33, 37]. The increased use of multi-core computers has made it beneficial to understand how to parallelize current solutions in order to make use of the available computing power and improve performance. The goal of most parallelized solutions has been to distribute the problem among several cores/processors ensuring that they are efficiently utilized in order to get close to a linear-speed up as the number of cores increases.

One of the factors that influences the parallel-algorithm design is the hardware architecture of the system. When designing a parallel algorithm to run on systems with shared memory and several cores or processors, the focus is usually to prevent simultaneous access to the shared memory by the working threads as this usually results in performance degradation. On the other hand, for parallel algorithms designed to run on traditional distributed systems where the computers are connected via a network and as such have distributed memory, the design focus is usually to minimize the message-passing overhead that occurs during the computation. In this thesis, we focus on work done on systems with shared memory and investigate the search-tree approach for CSP parallelization. This approach is currently the most promising method of achieving parallelization. Other approaches such as domain decomposition [19] which involves splitting the CSP into several easier sub-problems do not scale well.

3.2.1. Static Search Tree Distribution

Static search-tree-distribution methods of parallelizing CSPs involve modeling the problem as a tree and splitting it up into sub-trees *a priori*. The sub-trees are then assigned to different threads. In this thesis, we refer to the sub-trees as tasks. In this approach, each thread has the entire initial CSP problem and uses an existing sequential algorithm to solve the problem on a smaller search space. Research by Habbas et al. [20,

24], explored this approach in common CSP problems like the Langford and Golomb ruler problems [9, 20].

The search tree represents all possible combinations of values in the domains. A node in the tree represents a value of a variable and each level of the tree corresponds to a variable. The *l*-th level of the tree represents all the possible values of the *l*-th variable. Therefore, given a CSP with *n* variables, the height of the search tree is *n* and the values of variables traversed when going down the tree from the root node to the leaf node represent a potential solution to the CSP. Figure 3.1 shows an example of the search tree of a CSP with 3 variables *a*, *b*, and *c*. Assuming each variable has a domain size of *k*, then the cost of finding all solutions is at most the cost of visiting all nodes in the search tree which is $O(k^3)$.

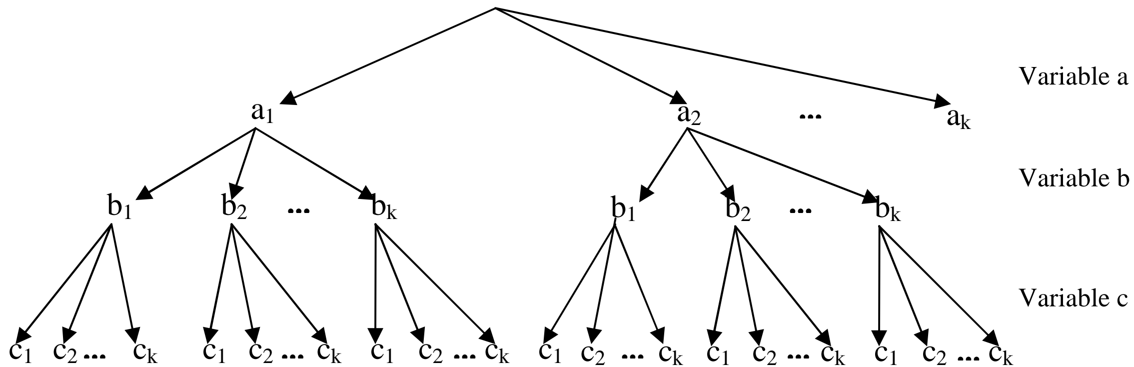


Figure 3.1: Search Tree of a CSP

Habbas et al. [20] proposed a generic method of generating tasks for parallelism from the search tree. In their method, they chose to explore the search tree up to a certain depth level *d*, which would result in up to k^d independent tasks (assuming again that the domain size of each variable is *k*). These tasks are then assigned to the processors using one of the various task-distribution strategies discussed next.

Task Distribution

In distributing the tasks among processors, the main challenge is load balancing, which ensures that all the processors are busy throughout the execution of the search algorithm. The difficulty with load balancing is that the work-effort involved in analyzing each task

usually varies. As such, the tasks are said to be *imbalanced*. More specifically, when traversing a particular sub-tree in a depth-first manner, it is possible that the constraints between variables at shallow points of the tree fail, in which case the backtracking algorithm does not need to go further down into the tree. On the other hand, in another sub-tree the algorithm may need to go deep into the tree before needing to backtrack. This results in an imbalanced work-effort when solving various tasks.

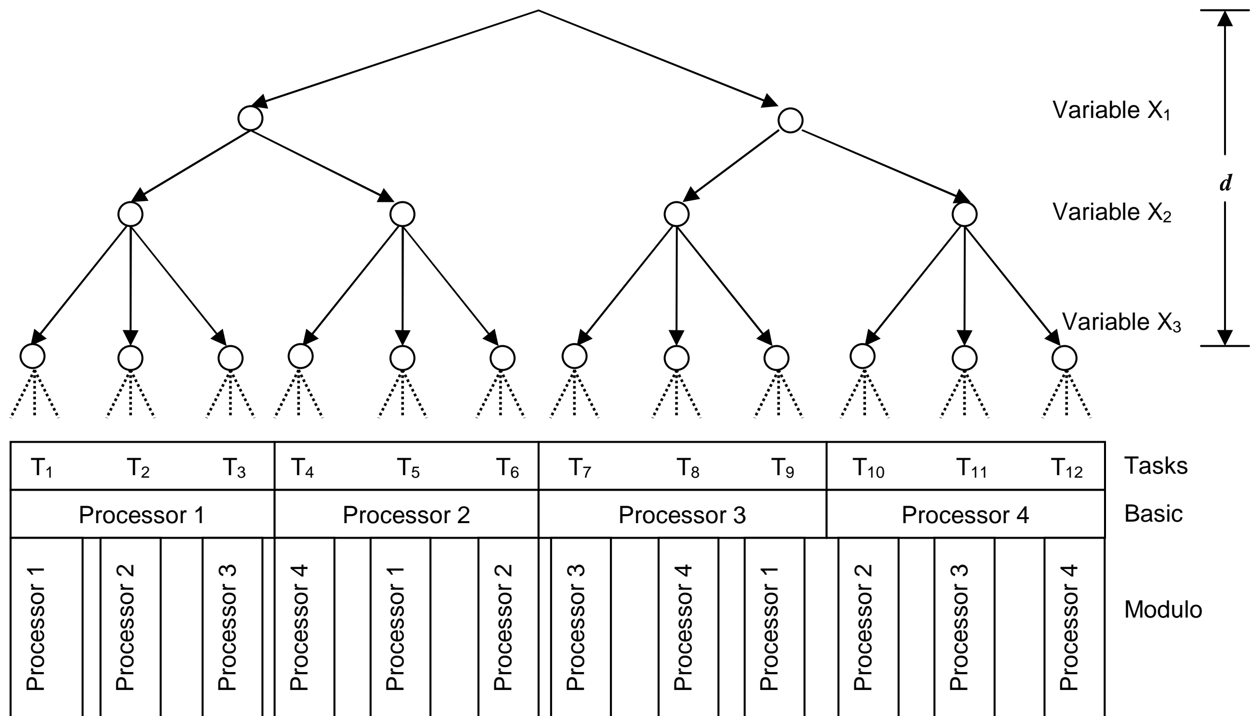


Figure 3.2: Task-Distribution Methods

There are various approaches to task distribution. The simplest method is the basic approach whereby the tasks are evenly distributed among the processors *a priori* (i.e., each processor gets *NumberOfTasks / NumberOfProcessors* tasks). The problem here is that it assumes the work-effort of each task is quite similar; as such this method usually results in poor performance for CSP problems with imbalanced sub-trees. Figure 3.2 shows the basic task-distribution method with four processors.

Another approach called “Modulo Number of Processors” aims at balancing the tasks among processors better and is shown in Figure 3.2. Here, task T_1 is assigned to Processor 1, T_2 to Processor 2 and continuing until all processors have been given a task and then the assignment is repeated beginning from the first processor [20]. In the third approach called dynamic task distribution, there is a server or shared resource holding all the tasks and each thread simply requests a new task when it has completed its current task. Habbas et al. compared the last two task-distribution approaches on the Langford’s problem and the dynamic approach performed better. In general, what approach works best depends on the particular CSP. Basic task-distribution methods incur less overhead but perform miserably for imbalanced search trees while dynamic approaches perform better, but with some overhead in task distribution.

3.2.2. Dynamic Work-stealing

Even with dynamic task-distribution strategies, it is still possible to have load-balancing issues where one processor is busy for an undesirable amount of time while the others are idle. Therefore, a lot of research has focused on dynamic work-stealing in which a processor can give away some of its work to an idle processor after the search has begun [14, 30, 33]. Obviously, this method incurs a lot more overhead and so it is important to have an efficient implementation when using any form of dynamic work-stealing in order to improve the overall performance of the algorithm. In Chapter 6, we develop a work-stealing strategy for the pattern-search problem in POET.

Chapter 4

Pattern-Search Parallelization

In this chapter we begin by representing the pattern-search problem in POET as a CSP and then we discuss the current sequential algorithm used. We then describe the algorithm used to parallelize the search.

4.1. Introduction

Re-written patterns in POET are represented in Conjunctive Normal Form (CNF) which is a set of disjunctions joined together by zero or more logical-AND operators. So for example, a pattern-search problem in POET given by ($a → $b ∨ $b → $c) ∧ ($c !→ $d) ∧ ($a → $c) is a CSP consisting of four variables $a, $b, $c and $d associated with event classes *A*, *B*, *C,* and *D* respectively. There are four constraints: $a happens-before $b, $b happens-before $c, $c does not happen-before $d, and $a happens-before $c. The set of relations here are all the values in event classes *A* and *B* that satisfy the first constraint, and all the values in *B* and *C*, *C* and *D*, and *A* and *C* that satisfy the second, third and fourth constraints respectively.

In POET, a solution to a pattern-search problem is an assignment of events to all or some of the variables in the pattern. POET allows the user to specify the set of variables for which values are to be returned. As discussed earlier, a variable marked with a tilde (~) implies that the primitive events associated with that variable should not be

returned to the user. Also, given a pattern that includes a variable marked by a universal quantifier (*), the search algorithm would only return events for the other variables not marked by the asterisk (or a tilde if present).

The pattern-matching algorithm used by POET is the naïve backtracking algorithm. Next, we discuss the details of this algorithm in relation to the pattern-search problem in POET and expand on the techniques discussed in Section 3.2.1 to achieve parallelism.

4.2. Naïve Backtracking Algorithm

Given a large set of primitive events belonging to various event classes and a pattern, the pattern-search problem involves finding the set of primitive events that satisfy the constraints in the pattern. For example, using the previous example (`$a → $b ∨ $b → $c`) ∧ (`$c !→ $d`) ∧ (`$a → $c`), the naive backtracking algorithm walks through the pattern from left to right assigning values to the variables and checking the constraints. If a constraint is satisfied, it moves on to the next constraint and assigns new values to unassigned variables. If a constraint is not satisfied with the current assignment, it picks the next value in that event class and keeps checking until a value that satisfies the constraint is found. If a value is not found, it backtracks to the variable that was last assigned and chooses the next value.

Therefore in this example, the algorithm initially starts out by looking at the first happens-before pair, (`$a → $b`), and assigns a primitive event from *A* to the variable `$a`, and a primitive event from *B* to the variable `$b`. It then checks if the event chosen from *A* happens before the event from *B*. Assuming that this constraint is satisfied, the algorithm skips over the second happens-before pair (since this is a disjunction, only one happens-before pair needs to be satisfied) and moves on to the only happens-before pair, (`$c !→ $d`), in the second disjunction. The algorithm then finds events from *C* and *D* that satisfy the constraint and moves on to the last happens-before pair (`$a → $c`). Since `$a` and `$c` have been previously assigned, the algorithm simply checks if their current values

satisfy the constraint. Assuming this is not the case, the algorithm backtracks to the previous happens-before pair ($c !→ $d) and chooses the next value for $d. The algorithm would have to exhaust all the values from *D* before going on to pick the next value for $c. It would then check for an event from *D* that happens after the new value of $c. When this occurs, the algorithm would move on to the last constraint hoping that the current value for $a now happens before the new assignment for $c.

Note that there are some inefficiencies with this algorithm, as on the first backtrack step, it would pick the next primitive event from *D* that satisfies the third constraint and then move to the last constraint repeating the precedence check with the same value assigned to $a and $c. It is only after it has backtracked to the third constraint several times and exhausted all the values from *D* for the current assignment of $c that it can pick the next value from *C* and move forward in the search. For now, we ignore this inefficiency and simply focus on the basic principles behind the search and how to achieve performance improvements through parallelization. Algorithm 4.1 shows a listing of the algorithm.

4.2.1. Cost Analysis

The backtracking algorithm is an exhaustive search that tries all variable-value combinations in searching for solutions that match the pattern. It is easy to see that the algorithm costs $O(k^n)$ where k is the maximum number of primitive events in an event class and n is the number of variables in the pattern.

Algorithm 4.1 : *Existing Naïve Backtracking Algorithm for Pattern Search*

```
1.  // Begin with the first disjunction
2.  position = 0
3.  disjunction = conjunction.list.get(position)
4.
5.  // Get the first happens-before pair in this disjunction
6.  hbpPos = 0
7.  hbpPair = disjunction.list.get(hbpPos)
8.  loop forever
9.      if hbpPair.first.isAlreadyAssigned
10.         firstValue = hbpPair.first.current()
11.     else
12.         firstValue = hbpPair.first.next()
13.         // No more events for this variable
14.         if firstValue is null break
15.     end if
16.
17.     loop forever
18.         if hbpPair.second.isAlreadyAssigned
19.             secondValue = hbpPair.second.current()
20.         else
21.             secondValue = hbpPair.second.next()
22.             if secondValue is null break
23.         end if
24.
25.         // Check if happens-before relationship is satisfied
26.         if isSatisfied(firstValue, secondValue)
27.             //store values
28.             matches.push(firstValue)
29.             matches.push(secondValue)
30.             if position == conjunction.list.size()
31.                 // No more disjunctions, end of search
32.                 return matches
33.             end if
34.             position = position + 1
35.             goto line 3
36.         else   // constraint not satisfied
37.             // Get next happens-before pair in the disjuction
38.             if hbpPos < disjunction.list.size()
39.                 hbpPos = hbpPos + 1
40.                 goto line 6
41.             end if
42.         end if
43.     end loop
44. end loop
```

4.3. Parallel Algorithm

Given a pattern-search problem in CNF, one approach to parallelizing this problem is to divide the pattern into several sub-problems and assign each sub-problem to a different thread (i.e., assign disjunctions to different threads). The partial solutions from each thread can then be combined into the final solution. Ideally, the sub-problems should be independent of each other, allowing each thread to work independently and avoid communication.

Unfortunately, it is difficult to create independent sub-problems as re-written patterns usually have a lot more constraints than variables, which makes it difficult to have sub-problems that do not share variables. Another problem is that it is difficult to estimate the work-effort of each sub-problem and so one sub-problem could be solved quickly while another could take a very long time. Such imbalance in work-effort could leave some threads busy and others idle, which does not fulfill one of the goals of parallelization. For the aforementioned reasons, it is reasonable to quickly conclude that dividing the pattern into sub-problems is not a promising approach. Therefore, we focus on employing the static search-tree-distribution methods as discussed in Section 3.2.1 in order to achieve parallelism.

In static search-tree distribution, the event-data set is divided into disjoint sub-trees or tasks that are assigned to the working threads. Each thread has the entire pattern and executes the naive backtracking algorithm on its own subset of the event data set. When a thread finds a solution it submits it to a master thread and then continues searching the current task for more solutions. Upon completion of the current task, the thread picks up another task. This process is repeated until all the tasks have been completed. The main issues we look at are how to divide the data set into tasks, the number of tasks to generate, and ensuring that memory access is localized. As mentioned before, the main goal is to keep the threads busy while minimizing the communication between threads in order to achieve a high level of parallelism.

4.3.1. Task Generation

In this thesis, we generate tasks by combining the search-tree distribution and task-distribution techniques described in the previous-work section. Given a search tree, we explore two approaches for task generation. The first approach which we call *grouped*, starts out by splitting the variables in the pattern into two sets. The first set is called *fixed* while the other set is called *unfixed*. Recall that the search-tree-distribution approach creates tasks by exploring certain variables up to a particular depth d of the search-tree. The *fixed* set consists of the variables that are initially explored and hence the size of this set is d. The remaining variables in the pattern make up the *unfixed* set. As discussed previously, the size of the tasks generated by this approach is at most k^d, where k is the size of the largest event class. In POET, the value of k could be over 100,000 events, and so this approach easily leads to an unmanageable number of tasks that consumes too much memory. There could also be a noticeable performance overhead as threads are synchronized when picking up their next task. To avoid this, we extend Habbas' approach by grouping the initial set of tasks into larger tasks. We start out with a pre-configured desired number of tasks S and group the initial set of tasks into approximately equal sizes in order to produce S tasks. Note that this approach is similar to the static method of task distribution, except that S is not always equivalent to the number of processors.

The *grouped* approach of task generation divides the search tree into relatively equal sub-trees; however, the work-effort involved in running the pattern search algorithm on each sub-tree usually differs, leading to idling threads. Since one of our goals is to keep all threads busy, we experiment with a variation of the "Modulo Number of Processors" task-distribution approach. We call this method *scattered* and here we start out by having S buckets and assign the first task to the first bucket, the second task to the next bucket, and so on, until the last bucket is filled. The process is repeated by assigning the next task to the first bucket until all tasks have been assigned to buckets. The "buckets" then represent the tasks that are assigned to each thread. Figure 4.1 illustrates the grouped and scattered task-generation methods.

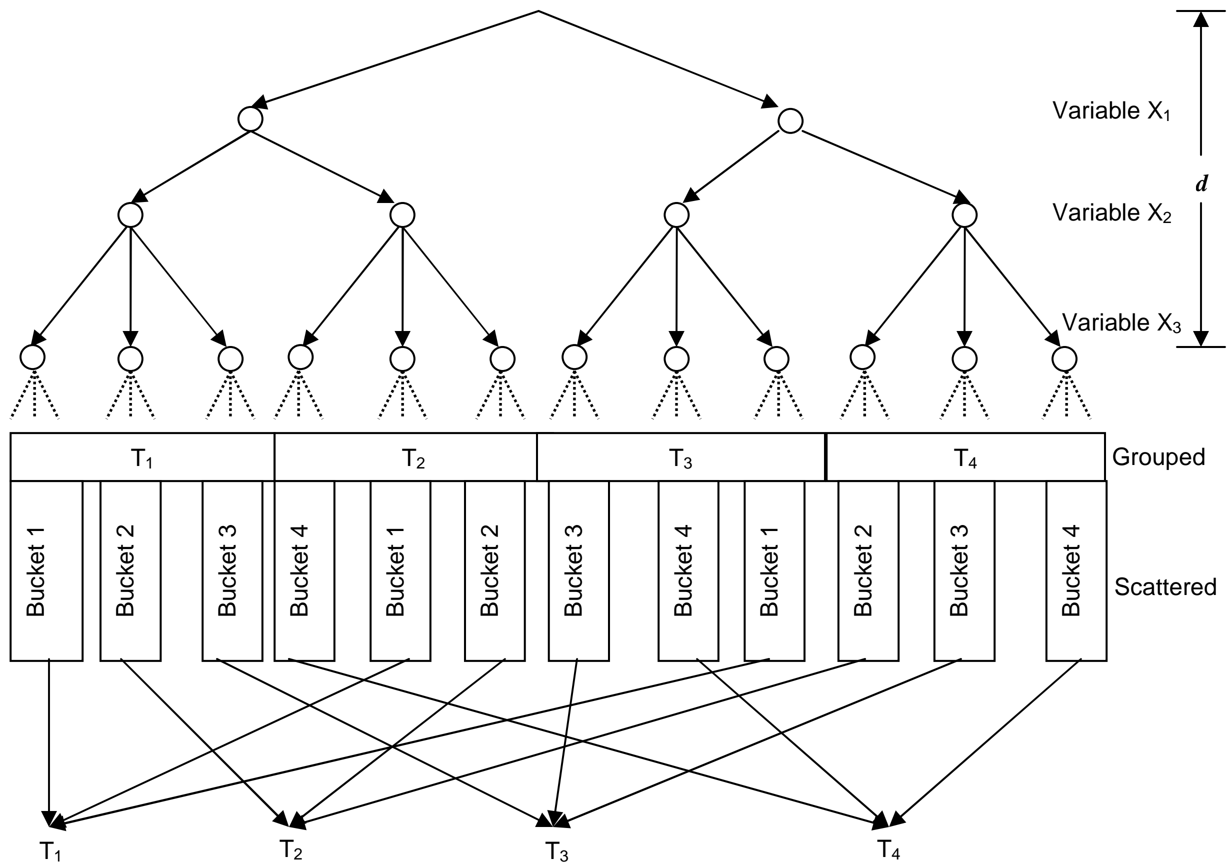


Figure 4.1: Grouped vs. Scattered Task-Generation Methods

In our experiments, we compare the two approaches and find that on average the grouped method performs better because each task maintains the ordering of events as they occurred in the target application. For most of the patterns used, maintaining this ordering in each task appears to be more favourable to the pattern-search algorithm than striving for more balanced tasks using the scattered approach. More details about the experimental setup and results are given in Chapter 5.

With the approaches discussed, certain questions arise, such as how to choose the *fixed* variables, and what are "good" values for S and d. In POET, Nichols [32] introduced a technique for re-ordering the variables in the pattern. The variable with the most constraints is placed at the beginning of the pattern. Using variable re-ordering resulted in better performance during the pattern-search. Therefore, we maintain this

ordering during parallelization by choosing the first *d* variables as the fixed variables after the pattern has been reordered.

In theory, a "good" value of *S* is one that is greater than the number of processors in order to ensure that there are always tasks available to keep the processors busy. In our experiments, *S* is set to the number of processors, twice the number of processors, four times the number of processors, and eight times the number of processors. Note that the cost of generating the tasks increases slightly as *S* increases, so it cannot be arbitrarily large. Finally, we begin by setting *d* to 1 and noting that at this point the number of tasks would be equivalent to the domain size of the first fixed variable. If this size is less than our desired number of tasks *S*, we simply increase the depth level until we can generate up to approximately *S* tasks.

There is a minor note about choosing the fixed variables when dealing with patterns that contain universal quantifiers. Recall that when a variable is qualified with a universal quantifier (*), then it means the constraint must be satisfied for all primitive events of the variable marked by the universal quantifier. Therefore, universally quantified variables cannot be fixed variables because each task must contain all the primitive events of the universally quantified variable.

4.3.2. Rules for Task Generation

The task-generation methods discussed involve taking the sub-trees at depth-level *d* and grouping them into larger-sized tasks. Certain rules must be followed when creating these groups in order to avoid "duplicate work-effort" when the backtracking algorithm is executed. Here, "duplicate work-effort" refers to the scenario where more than one thread visits the same set of nodes (i.e., primitive event values) from a top level node to a leaf node in the search tree. Note that the nodes here represent the primitive event values.

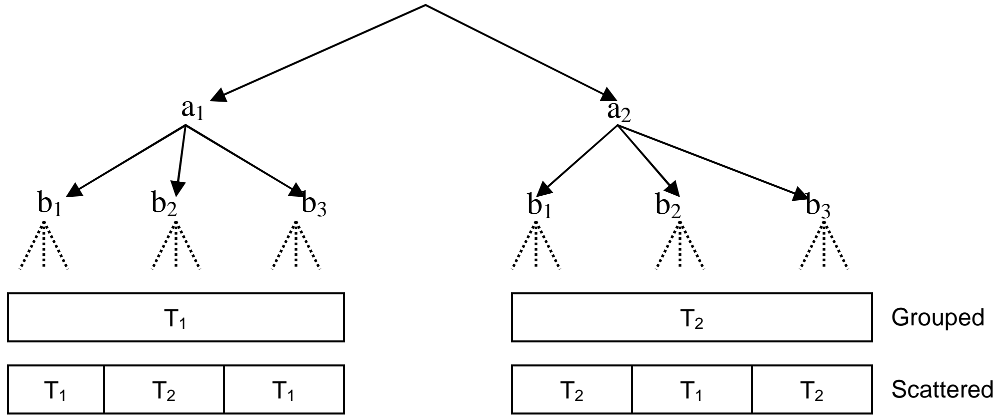


Figure 4.2: Task-Generation Example
(Grouped: Independent Tasks, Scattered: Duplicate Work)

For example, Figure 4.2 represents the search tree up to the second depth-level for a pattern consisting of three variables `$a`, `$b`, and `$c` belonging to event classes *A*, *B*, and *C*. The search-tree portion for variable `$c` is not shown in the figure. The figure shows that event class *A* contains two primitive events a_1 and a_2, while that of *B* contains three events b_1, b_2, and b_3. If the pre-defined number of tasks to be generated, *S*, is 2, then a valid set of tasks using the grouped approach is shown in the Table 4.1. Note that with these tasks, when the backtracking algorithm is run, the sub-tree of task T_1 is completely independent of T_2 as the thread assigned to work on task T_1 will never visit a node that is already visited or will be visited by the other thread working on task T_2.

Table 4.1: Grouped Task Generation Example (Independent Work)

Tasks / Variables	**$a**	**$b**	**$c**
T_1	a_1	b_1, b_2, b_3	All events in *C*
T_2	a_2	b_1, b_2, b_3	All events in *C*

In the scattered approach, the two tasks end up consisting of the entire search tree so both threads end up repeating the exactly the same amount of work. Table 4.2 shows the set of tasks in the scattered approach.

Table 4.2: Scattered Task Generation Example (Duplicate Work)

Tasks / Variables	$a	$b	$c
T_1	a_1, a_2	b_1, b_2, b_3	All events in *C*
T_2	a_1, a_2	b_1, b_2, b_3	All events in *C*

Now, assuming *S* is 3, the grouped and scattered methods will generate the tasks as shown in Figure 4.3.

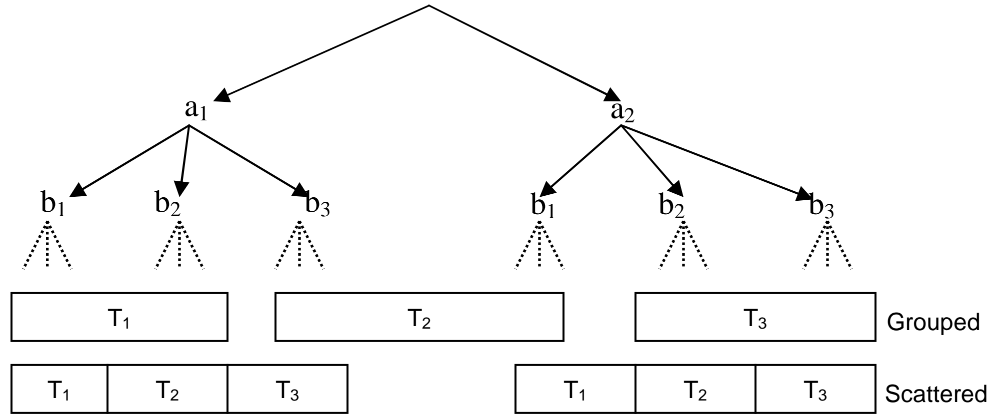


Figure 4.3: Task-Generation Example
(Grouped: Duplicate Work, Scattered: Independent Work)

Following the figure, in the grouped approach, the primitive events will be distributed into the three tasks as shown in Table 4.3. In this table, different threads working on tasks T_1 and T_2 will end up both exploring values a_1 and b_1. Also, the threads working on tasks T_2 and T_3 will end up both exploring values a_2 and b_3. Obviously, splitting the search tree in this manner will result in duplicate work-effort thereby degrading the performance of the parallel algorithm. Using 3 tasks for the scattered approach does not result in duplicate work in this case. The examples show that for both approaches, duplicate work-effort is possible when generating the tasks.

Table 4.3: Grouped Task Generation Example (Duplicate Work)

Tasks / Variables	$a	$b	$c
T_1	a_1	b_1, b_2	All values in C
T_2	a_1, a_2	b_1, b_3	All values in C
T_3	a_2	b_2, b_3	All values in C

In order to avoid such duplicate effort when using either approach for task-generation, we follow certain rules when generating tasks. Note that these rules are generic enough to be applied to any parallel search-tree implementation of CSPs with a large domain space that employs the backtracking algorithm. Previous work mainly looked at eliminating useless tasks when it has been determined that a path up to a certain depth-level has failed. As such, any other tasks beginning with those initial fixed variables can be skipped [24]. In our approach, we focus on eliminating duplicate work that could exist in a task when the backtracking algorithm is executed. In addition, these rules are applied during task-generation and not when the threads have started working. First, we propose the following definitions that simplify the discussion of the rules.

Definition 4.1:

A *fixed* node in the search-tree is a node associated with a fixed variable. For example, from Figure 4.3, the nodes with values a_1, a_2, and b_1 are fixed nodes.

Definition 4.2:

A *fixed leaf* node is a fixed node at depth-level d of the search tree.

Definition 4.3:

An *ancestor* node is any fixed node at level 1 to level d - 1. So fixed leaf nodes are not considered ancestor nodes.

Definition 4.4:

An ancestor node of a given node n_1 at level l, ($1 < l \leq d$) is any fixed node along the path to n_1 from level 1 to l - 1.

Definition 4.5:

A node n_1 is a *perfect sibling* of another node n_2 if n_1 and n_2 are on the same level of the search-tree and each have child nodes with exactly the same set of values. Note that when

looking at an entire search-tree, all the nodes on the same level are perfect siblings of each other. This is not necessarily the case for sub-trees as we will see shortly.

Definition 4.6:

A node n_1 is an *imperfect sibling* of another node n_2 if n_1 and n_2 are on the same level of the search-tree with each having a set of values C_1 and C_2 respectively associated with their child nodes and there exists a value in C_1 that is not in C_2, or vice versa. In other words, n_1 and n_2 do not have exactly the same set of values associated with their child nodes. Imperfect siblings only occur when looking at sub-trees.

Given a task T_i, let F_i represent the set of values of the fixed nodes in T_i and let F_{il} represent the set of values for the fixed nodes in T_i at level l.

Definition 4.7:

Two tasks T_i and T_j developed with sub-trees up to depth level d are disjoint if there exists a level l, $(1 \leq l \leq d)$, where the intersection of the sets of values of the fixed nodes at that level is empty i.e., $F_{il} \cap F_{jl} = \emptyset$.

With the above definitions, one rule we need to apply when generating tasks in order to avoid duplicate work when the backtracking algorithm is executed is as follows:

Rule 4.1:

In the set of tasks generated from the entire search tree, each task must be disjoint from every other task.

The following theorem proves that if Rule 4.1 is followed then it is not possible to have duplicate work.

Theorem 4.1:

If two tasks T_i and T_j are disjoint then the threads P_i and P_j assigned to tasks T_i and T_j respectively will never visit exactly the same set of primitive events from level 1 to n of the search tree, where n is the height of the tree.

Proof: We prove this by contradiction. Assume the tasks T_i and T_j are disjoint but it is possible for P_i and P_j to visit the same set of primitive events from level 1 to n, then it means that at each level l in T_i and T_j there is a primitive event e that is present in F_{il} and F_{jl} that can be visited by both threads. This implies that $F_{il} \cap F_{jl} \neq \emptyset$, $\forall l \in \{1, ..., n\}$, and as such the tasks T_i and T_j cannot be disjoint based on Definition 4.7. Therefore, it must be the case if T_i and T_j are disjoint, then P_i and P_j cannot visit the same set of primitive events from level 1 to n, which implies there is no duplicate work. ■

Based on the premise that the set of tasks for a search tree covers the entire tree (i.e., there is no missing work), the next theorem postulates that if a sub-tree representing a task has certain properties then it is possible to have duplicate work.

Theorem 4.2:

If a task T_i has fixed leaf nodes with values z_1 and z_2 (where z_1 and z_2 can be equal) that have ancestor nodes with different values h_1 and h_2, respectively, that are imperfect siblings, then there must exist another task T_j from the search tree such that T_i and T_j are not disjoint.

Proof: Since the ancestor nodes with values h_1 and h_2 are imperfect siblings, in task T_i either there is a node with primitive event value e that is a child of the ancestor node with value h_1 but not a child of the ancestor node with value h_2 or there is a node with primitive event value f that is a child of the ancestor node with value h_2 but not a child of the ancestor node with value h_1 (Definition 4.6). We prove only one case as the proof for the other case is similar.

Case 1: We consider the case where in T_i there is a node with primitive event e that is a child of the ancestor node with value h_1 but not a child of the ancestor node with value h_2. To show that T_i and T_j are not disjoint, we simply need to show that $F_{il} \cap F_{jl} \neq \emptyset$, $\forall l \in \{1, ..., d\}$. Let the path with node values $n_{i1} \rightarrow n_{i2} \rightarrow .... \rightarrow (n_{im} = h_1) \rightarrow (n_{i(m+1)} = e) \rightarrow n_{i(m+2)} \rightarrow .... \rightarrow (n_{id} = z_1)$ be present in T_i. It is easy to see that there must exist another task T_j that contains the path with node values $(n_{j1} = n_{i1}) \rightarrow (n_{j2} = n_{i2}) \rightarrow .... \rightarrow (n_{j(m-1)} = n_{i(m-1)}) \rightarrow (n_{jm} = h_2) \rightarrow (n_{i(m+1)} = n_{j(m+1)} = e) \rightarrow (n_{j(m+2)} = n_{i(m+2)}) \rightarrow .... \rightarrow (n_{jd} = n_{id} = z_1)$ as each node on each level in the search tree contain exactly the same child node values (i.e., are perfect siblings of each other) and all tasks combined cover the entire search tree (i.e., there is no missing work). Note that on the m-th level, we already know that h_2 is in T_i and T_j so $F_{im} \cap F_{jm} \neq \emptyset$. Therefore, in tasks T_i and T_j, $F_{i1} \cap F_{j1} = n_{i1}$, $F_{i2} \cap F_{j2} = n_{i2}$, ..., $F_{i(m-1)} \cap F_{j(m-1)} = n_{i(m-1)}$, $F_{im} \cap F_{jm} = h_2$, $F_{i(m+1)} \cap F_{j(m+1)} = e$, $F_{i(m+2)} \cap F_{j(m+2)} = n_{i(m+2)}$, ..., $F_{id} \cap F_{jd} = z_1$ and hence T_i and T_j are not disjoint. ■

Taking note of the above theorem, we employ the following rule when generating tasks.

Rule 4.2:

During task-generation, there should not be any task that has a pair of fixed leaf nodes with ancestor nodes (with different values) that are imperfect siblings.

Next, we propose the following lemmas that will enable us to prove that our task-generation implementation (described in the next section) ensures that no duplicate work-effort occurs.

Lemma 4.1:

There is a unique path from level 1 to each node at any level l in the search-tree, where $1 \leq l \leq n$ (n is the last level in the search-tree).

Proof: This can be easily verified based on the way the search tree is constructed. ■

Lemma 4.2:

Given two unique paths p_i and p_j from levels 1 to l in a search-tree, there exists a level m ($1 \leq m \leq l$) such that the primitive event values representing the nodes at n_{im} and n_{jm} are not equal.

Proof: Again, this can be easily verified based on the search-tree structure. ■

4.3.3. Task-Generation Implementation

In this section, we describe the algorithms used for generating tasks in the grouped and scattered approaches. Each algorithm takes as input a desired number of tasks to be generated, *S*, and based on the search-tree generates a set of tasks whose size is as close to *S* as possible. The goal is to generate a number of tasks as close to *S* as possible without violating Rules 4.1 and 4.2.

Recall that the total number of tasks possible at depth level *d* is given by $|D_1| * |D_2| * ... * |D_d|$, where $|D_i|$ is the domain size of the variable at level *i*. Using this information, we can determine the depth level needed in order to generate up to *S* tasks by simply going down one more level if the current total number of tasks possible is not up to *S*. Now, given a certain depth level *d*, we describe each algorithm for task generation and prove that the algorithms avoid duplicate work-effort among the threads.

Grouped Approach

I. ***Case 1*****:** If *d* is 1, then we simply divide the total number of nodes at this level (we call this *M*), by *S* to get the number of nodes in each group (otherwise called the group size), *g*. The algorithm then assigns the nodes at level 1 from positions 1 to *g* to task T_1, positions $g + 1$ to $2g$ to task T_2, positions $2g + 1$ to $3g$ to task T_3, ..., positions (C-1) $g + 1$ to Cg to task T_{S-1}, and positions Cg + 1 to *M* to task T_S.

It is easy to see that the sets of tasks generated above are disjoint from each other as the nodes in level 1 are unique and there is no overlap of nodes during the task assignment. Therefore, $F_{i1} \cap F_{j1} = \emptyset$, $\forall i, j \in \{1, ..., S\}$, so there is no duplicate work-effort.

Case 2**:** If d is greater than 1, we again set the group size, g, to the ceiling of M / S, and then we adjust this value to a new value g' in order to generate a set of relatively equal-sized tasks whose size is as close to S as possible. To understand why this adjustment is necessary, we first note that to enforce Rule 4.2 the algorithm creates a new task when it gets to the first leaf node under the next sub-tree rooted at level d -1. This implies that the last task generated from the previous sub-tree may not contain up to g' fixed leaf nodes. This restriction is needed in order to avoid generating tasks with fixed leaf nodes that have different ancestor nodes that are imperfect siblings (see Figure 4.4).

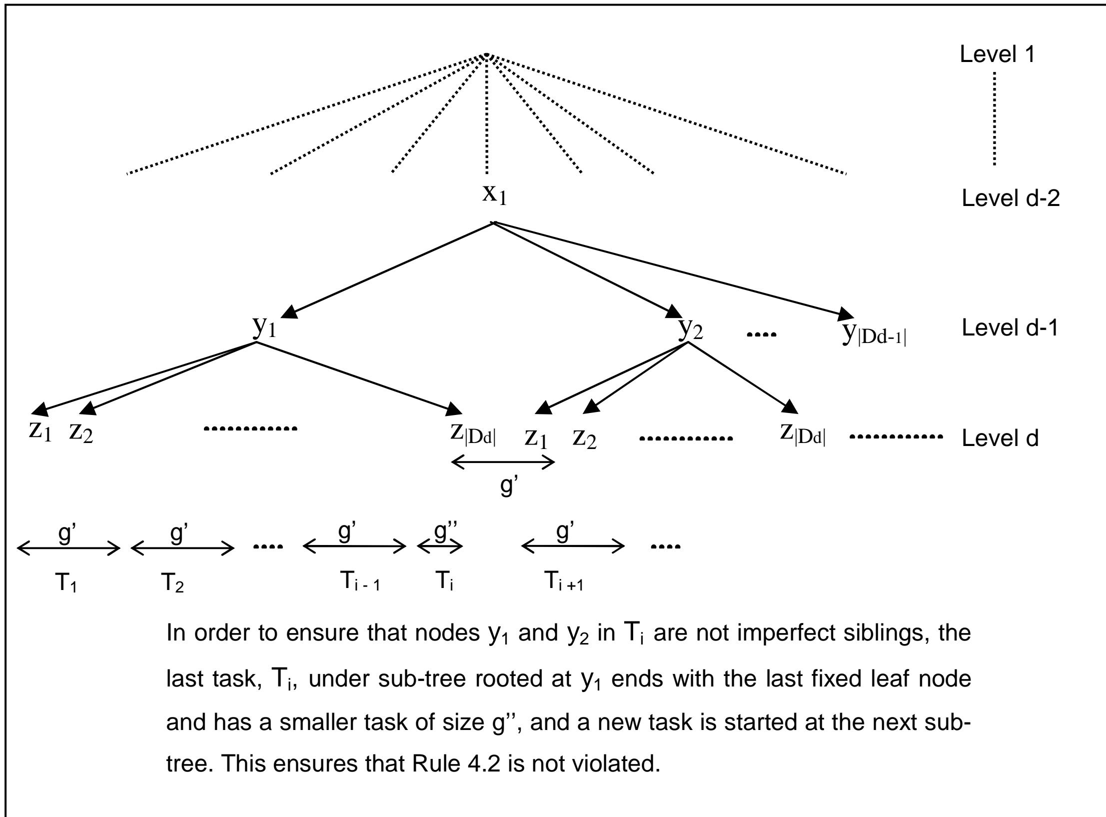


Figure 4.4: Grouped Approach – Enforcing Rule 4.2

Without adjusting the group size, we note two problems that occur as a result of this restriction. Firstly, we see that if g is greater than half of $|D_d|$, there will be exactly two tasks generated under each sub-tree rooted at level d -1 as the algorithm generates a new task starting at the following sub-tree. This produces a set of tasks whose size is almost two times the value of S in the worst case. As mentioned previously, having too many tasks could reduce the efficiency of the parallel algorithm. Secondly, we see that exactly half of the set of tasks generated would potentially have a much smaller size than the other half. Having too many unequal-sized tasks increases the likelihood of introducing a work imbalance among the threads which may have been avoided with relatively even-sized tasks (though there is no guarantee of this). To avoid the aforementioned problems, when g is greater than half of $|D_d|$, we adjust g using the following technique:

> Let λ represent a number between 0.5 and 1 (exclusive) such that $\lambda * |D_d|$ is a threshold value. When g is greater than this threshold, it is adjusted to $|D_d|$; otherwise it is adjusted to half of $|D_d|$. Note that higher values of λ result in a set of tasks whose size is much greater than S.

In our implementation, λ was set to 0.707 which ensures that the actual size of tasks generated, denoted by S', is at most 1.414 times the initial desired size, S (see Appendix A for details on the relationship between λ, S and S'). Note that when g is less than half of $|D_d|$, it is not adjusted because though we can still have uneven-sized tasks, the number of tasks with equal sizes will be more than the number of tasks with unequal sizes. In fact, it is easy to see that in the worst case, at least 66% (i.e., at least 2 under each sub-tree, hence 2 of 3 tasks) of the tasks generated will have equal sizes. Also, with the grouped approach, it is impossible for the initial group size, g, to be greater than $|D_d|$ as this implies that there was no point going as far down as level d in the search tree in order to generate the desired number of tasks. Having discussed why and how the group size is adjusted, we move on to

proving that the tasks generated using this grouped approached are always disjoint from each other.

Proof (Case 2a): If $g' \leq 0.5|D_d|$, then there are two or more tasks whose fixed leaf nodes have the same parent node at level $d - 1$. Consider any two tasks T_i and T_j, if their set of fixed leaf nodes both have the same parent node, then we can easily conclude that $F_{id} \cap F_{jd} = \varnothing$ as there are no overlapping values when assigning fixed leaf nodes to a task.

If the set of fixed leaf nodes in T_i and T_j do not have the same parent node then either both tasks have no common value in their set of fixed-leaf-node values i.e., $F_{id} \cap F_{jd} = \varnothing$ or both tasks contain exactly the same set of values associated with their fixed leaf nodes, i.e., $(F_{id} \equiv F_{jd}) \subset D_d$. This is because the parent nodes are perfect siblings and the algorithm always starts a new task when it gets to the first fixed leaf node under a new sub-tree and continues the task-generation with a group size of g'. Therefore, in the former case when $F_{id} \cap F_{jd} = \varnothing$, the tasks T_i and T_j are disjoint based on Definition 4.7.

We now show that when $(F_{id} \equiv F_{jd}) \subset D_d$ (the latter case), the algorithm ensures that T_i and T_j are disjoint. According to Lemmas 4.1 and 4.2, we know that there is a unique path to each of the parent nodes of the set of fixed leaf nodes in T_i and T_j, as such there is a level m with different node values on each of these paths. Therefore, T_i is disjoint from T_j at level m ($F_{im} \cap F_{jm} = \varnothing$). ■

***Proof (Case 2b)*:** If $g' = |D_d|$, then any two tasks T_i and T_j have exactly the same set of fixed leaf nodes, i.e., $F_{id} \cap F_{jd} \equiv D_d$ and different parent nodes. We see that the proof is the same as the latter case of second scenario in *Case 2a* above and so T_i and T_j are disjoint. Therefore, there is no duplicate work-effort based on Theorem 4.1. ■

Scattered Approach

I. ***Case 1*****:** If d is 1, the algorithm simply assigns the nodes (at level 1) at position 1 to task T_1, position 2 to task T_2, position 3 to task T_3, ..., position S to T_S, and then repeats the process by assigning the node at position $S + 1$ to task T_1. Since all the nodes at each level in the search tree have unique values then it follows that T_i and T_j are disjoint as $F_{i1} \cap F_{j1} = \emptyset$, $\forall i, j \in \{1, ..., S\}$. Therefore, there will be no duplicate work-effort.

II. ***Case 2:*** If $d > 1$ and $S \leq |D_d|$, the algorithm generates the tasks as follows. For similar reasons discussed previously, we first adjust S in exactly the same manner as the group size in the grouped approach to get the new task size S'. The algorithm then assigns the fixed leaf nodes at positions 1 to S' to the buckets B_1 to $B_{S'}$ associated with tasks T_1 to $T_{S'}$. It then repeats the process again filling bucket B_1 at the next fixed leaf node at position $S' + 1$. In order to enforce Rule 4.2, we ensure that the first fixed leaf node under a new sub-tree rooted at level d -1 is assigned to bucket B_1. This ensures that the set of fixed leaf nodes assigned to each task remains the same as the algorithm progresses (see Figure 4.5).

Now we prove that the algorithm guarantees that the tasks generated are disjoint from each other.

Proof: As noted before, the algorithm ensures that each task T_i has exactly the same set of fixed leaf nodes. Therefore, task T_i is disjoint from another task T_j as $F_{id} \cap F_{jd} = \emptyset$, $\forall i, j \in \{1, ..., S'\}$ and so there will be no duplicate work-effort. ■

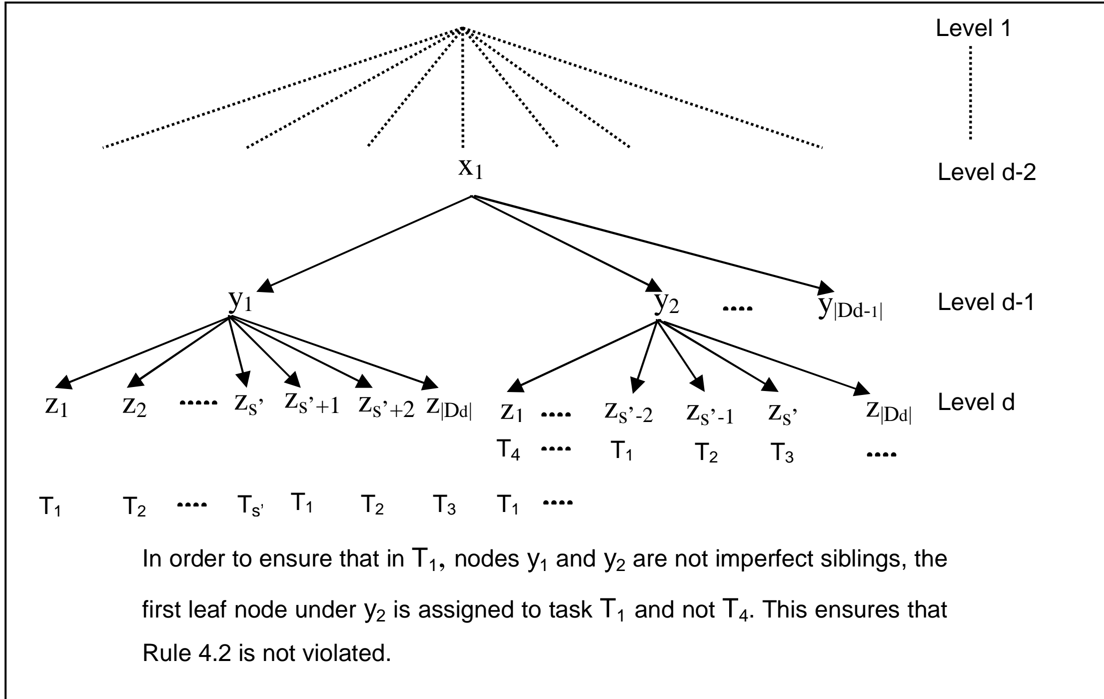


Figure 4.5: Scattered Approach – Case 2 ($S \leq |D_d|$)

Case 3: If $d > 1$ and $S > |D_d|$, the algorithm generates the tasks as follows. First, we adjust S according to Algorithm 4.2 to get the new task size S'. This algorithm sets S' to the total number of fixed leaf nodes (closest to S) under a sub-tree. In this case, this adjustment is even more important in order to avoid duplicate work-effort as we will see in the proof shortly. The algorithm then fills the buckets B_1 to $B_{S'}$ as in Case 2. We now prove that this algorithm ensures that there is no duplicate work-effort.

Algorithm 4.2: *Adjusting the desired number of tasks (S)*

1. // Initialize variables
2. S' = 0, Y = | D_d |
3. // Loop through each level moving towards the root of the tree
4. **for** i = (d - 1) ... 1
5. Y' = Y
6. Y = Y * | D_i |
7. **if** S ≤ Y **break**
8. **end**
9. // The method closerTo would return the number Y or Y' that is closer
10. // to S
11. S' = closerTo(S, Y', Y)

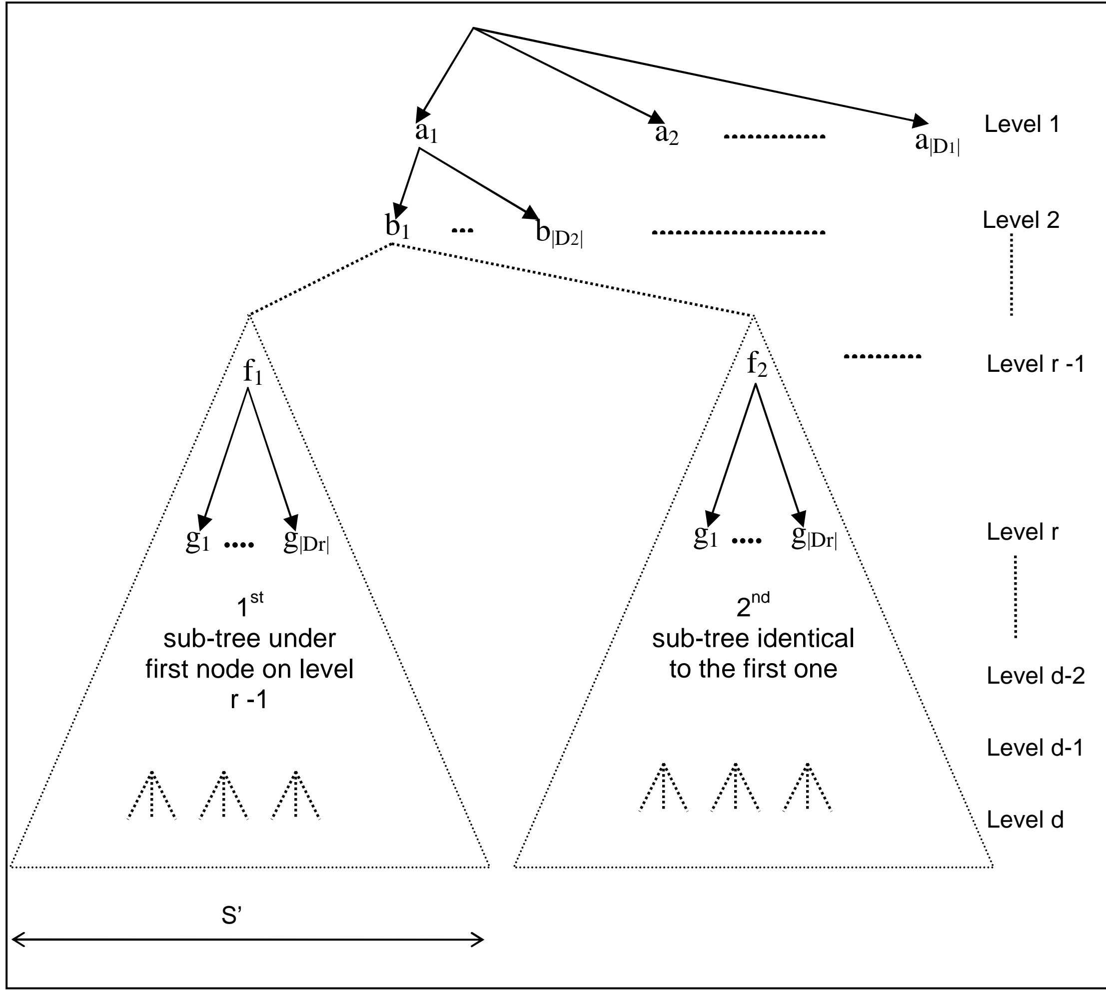


Figure 4.6: Scattered Approach – Case 3 ($S > |D_d|$)

Proof: If $S' = |D_d|$ then the proof is same as Case 2 above. If $S' > |D_d|$ then $S' = |D_d| * ... * |D_r|$, $1 \leq r \leq d - 1$ based on Algorithm 4.2. If we consider any two buckets B_i and B_j associated with tasks T_i and T_j in the first sub-tree rooted at the first node on level $r - 1$ (see Figure 4.6), we see that this sub-tree contains the first paths p_i and p_j that are assigned to the buckets B_i and B_j respectively. From Lemma 4.1, we know that p_i and p_j are unique and so according to Lemma 4.2, there is a level m, ($m \geq r$), where there are unequal nodes on each of these paths. As the algorithm moves through each node rooted at level r - 1 and fills the buckets associated with T_1 to $T_{S'}$ again, we note that the next paths assigned to each bucket will each contain the same sequence of nodes from level r to d since all the sub-

trees rooted at level r - 1 are identical. Therefore, T_i and T_j are disjoint at level m ($F_{im} \cap F_{jm} = \emptyset$) and so there is no duplicate work-effort, based on Theorem 4.1. ■

4.3.4. Memory Localization

As previously mentioned, this thesis focuses on parallelization on a multi-core or multi-processor machine with shared memory. As such, in order to avoid the huge performance degradation that occurs when multiple threads access the same memory location, we create local objects that are accessed by each thread while the search is running.

Algorithm 4.3: *Creating a copy of the Conjunction object for each task*

```
1.  // Create a new Conjunction object containing the same number of
    // Disjunction, HappensBeforePair and  Factor objects.
2.  Conjunction  newConj = origConj.copy()

3.  // Lines 4 and 5 are the results of the task-generation step
4.  fixedFactors = getFixedFactors()
5.  unfixedFactors = getUnfixedFactors()
6.  for i = 0 ... origConj.list.size()        //Number of disjunctions
7.     disj = origConj.list.get(i)
8.     newDisj = newConj.list.get(i)
9.     for j = 0 ... newDisj.list.size()   //Number of happen-before pairs
10.       hbp = disj.list.get(j)

11.       for k = 0 ... fixedFactors.size()
12.          if fixedFactors.get(k) is hbp.first
13.             newDisj.list.get(j).first = fixedFactors.get(k).copy()
14.         else if fixedFactors.get(k) is hbp.second
15.             newDisj.list.get(j).second = fixedFactors.get(k).copy()
16.         end if
17.       end for
18.       for k = 0 ... unfixedFactors.size()
19.          if unfixedFactors.get(k) is hbp.first
20.             newDisj.list.get(j).first = unfixedFactors.get(k).copy()
21.         else if unfixedFactors.get(k) is hbp.second
22.             newDisj.list.get(j).second = unfixedFactors.get(k).copy()
23.         end if
24.      end for
25.
26.    end for
27. end for
```

In the POET class structure, the *Conjunction* object contains the pattern in CNF form. As such it consists of a list of *Disjunction* objects and the constraints between variables are represented by a *HappensBeforePair* object. Each *HappensBeforePair* object contains two variables, each represented by a *Factor* object that holds the list of *PrimitiveEvent* objects (representing the primitive events) associated with the variable. Each task generated has a separate "deep" copy of the *Conjunction* object and separate lists of primitive events. Though the *PrimitiveEvent* objects associated with the unfixed variables would be shared by various threads, we avoid copying these objects as they are read-only and do not require thread synchronization. Algorithm 4.3 shows the steps for creating a copy of the *Conjunction* object that contains a subset of the search tree.

Another object that was localized is a map that contains the timestamp cache. POET maintains a list of vector timestamps for each primitive event. This cache is loaded into memory before the search algorithm runs and is used to determine the precedence relationship between two variables. Therefore, to avoid performance degradation, each thread is given a shallow copy of the timestamp cache. Since we are not making copies of the timestamp objects, the memory cost here is negligible.

4.3.5. Cost Analysis of Task Generation

As discussed previously, the cost of task generation is equivalent to the cost of traversing the search tree up to a certain depth level d, i.e., $O(k^d)$, where k is the size of the largest event class. Therefore, it is important to keep d reasonably low in order to reduce the cost of task generation. Another additional cost is the cost of creating copies of the *Conjunction* object and the timestamp cache. The cost of making a copy of a *Conjunction* object depends on the number of disjunctions, l, the number of happens-before pairs (i.e., the constraints), m, and the number of variables, n. The cost of going through the outer for-loop is l, the cost of the first nested for-loop is m, and the cost of the two innermost for-loops is n (i.e., going through the fixed and unfixed variables). Thus the total cost is $O(l * m * n)$. From the experiments, we see that the cost of creating the tasks is

insignificant compared to the cost of running the search algorithm, so it is usually beneficial to run the parallel algorithm over the sequential one.

4.4. Optimization for Universal Quantifiers

Nichols [32] describes an optimization to the search algorithm for patterns containing universal quantifiers, since evaluating such patterns can be very time-consuming. In evaluating the pattern ($a → *b), and assuming that $a and *b are associated with event classes *A* and *B*, the search algorithm would pick a value for $a from *A* and check if this value happens-before each value of *b. If the algorithm finds that this value of $a does not happen-before a value of *b, then it moves on to the next value for $a and repeats the search beginning with the first value of *b. Nichols hypothesized that due to the ordering of primitive events, it is more likely that the next value of $a will fail at the same point or a point close to the value of *b which caused the previous value of $a to fail. Based on this hypothesis, Nichols proposed that the value of *b that failed should be moved to the start of the list of events so that when the next value of $a is picked, the precedence check will fail earlier. Experiments showed that this re-ordering of the primitive events associated with the universally quantified variable gave a huge performance boost for evaluating certain patterns.

Though this optimization works well in the sequential algorithm, its benefit is not fully realized when running the proposed parallel algorithm. This is because the task-generation phase divides the list of events into various subsets which are handled by different threads, so this optimization becomes localized to each task. Following the previous example, when a value of *b is moved to the start of the event list, only the thread handling this task "sees" the benefit by quickly failing on the next value of $a. When the thread picks up its next task, it loses this information and begins with the first value of $a and the first value of *b in its new sub-tree search space.

In order to improve the parallel algorithm when running patterns with universally-quantified variables, we propose certain changes to the task-distribution algorithm. To

simplify the discussion of these changes, let $Y = \{*y_1, *y_2, ..., *y_n\}$ be the set of universally quantified variables in the pattern and $Z = \{Z_1, Z_2, ..., Z_n\}$ be the set of domains for these universally quantified variables, i.e., each Z_i represents the sequence of primitive events associated with variable $*y_i$. Now, after a thread completes a task, let Z_i' represent the new sequence of primitive events for variable $*y_i$ that is produced due to the optimization for universally quantified variables proposed by Nichols. The changes to the task-distribution algorithm are as follows:

1. First, we divide the set of |T| tasks into |T| / *t* subsets, where *t* is the number of threads. Each thread is then assigned a subset of the tasks.
2. Each thread begins by selecting a task from its own subset and running the backtracking algorithm as usual.
3. When a thread completes a task, it selects the next task in its subset (or if there is not one, it would select the next available task from another thread's subset), and for each universally quantified variable $*y_i$, the thread would set the variable's domain Z_i to Z_i'. The thread would then proceed to run the backtracking algorithm on this task using the re-ordered set of domains.

Note that because this optimization relies on the order of the primitive events in each domain, it naturally favours the *grouped* task-generation approach. Section 5.2, evaluates the benefits of this optimization for patterns containing universal quantifiers.

4.5. Parallel Pattern-Search Architecture

In this section, we describe the overall architecture of the parallel pattern-search feature, the co-ordination among threads, and how this new feature fits into the existing POET pattern-search application.

Some parameters that can be used to configure the parallel algorithm were added into the existing *poet.properties*. The first parameter, *poet.core.searcher.mode*, is used to specify whether to run the sequential algorithm and return one match at a time (value:

“*default”*), or to run the parallel algorithm and return one match at a time (value: *“parallelOneMatch”*), or to run the sequential algorithm and return all matches at once (value: *“nonParallelAllMatches”*), or to run the parallel algorithm and return all matches at once (value: *“parallelAllMatches”*).

The next parameter, *poet.core.searcher.parallel.subtaskfactor* (otherwise called sub-task factor), is used to determine how many tasks to generate and is specified as a factor by which the number of processors/cores should be multiplied. The default value is eight meaning that the number of tasks the algorithm should generate is eight times the number of cores. The *poet.core.searcher.parallel.numthreads* parameter specifies the number of threads to use and the default value is equivalent to the number of cores available. The *poet.core.searcher.parallel.taskcreationtype* parameter is used to specify the technique to be used for task-generation, i.e., the *grouped* or *scattered* approach.

In POET, the pattern-search application begins by parsing a predicate file containing the list of patterns. It then populates the timestamp cache and loads the primitive events into memory. The user then selects a pattern and the search algorithm is invoked. Once a match is found, it is displayed to the user and the user can select a button in order to retrieve the next match. When running the sequential algorithm, displaying a match to the user is achieved using two threads. The main thread initializes the pattern-search class, *FindFlattenedMatch.java*, and creates a search thread that begins the pattern-matching algorithm. The main thread then goes to sleep on a semaphore. When the search thread finds a match, it wakes up the main thread and then goes to sleep. The main thread then displays the match to the user, and if the user requests another match, the search thread will be woken up to continue the search from where it left off.

In the parallel mode (i.e., *parallelOneMatch*), we achieve a similar execution sequence by making use of several threads and counting semaphores. The main thread initializes the pattern-search class, *ParallelMatchFinder.java*, and creates a master thread to begin the search. The main thread then goes to sleep in a similar manner. The master thread starts out by splitting the variables in the given pattern into *fixed* and *unfixed* sets. The thread then generates the tasks, with each task containing a copy of the pattern and two counting semaphores. The first semaphore, called the master semaphore, is used by the master thread and each task has a reference to this semaphore. The second semaphore

is used only by the thread executing the task. The master thread then creates the required number of searcher threads and goes to sleep on its semaphore. The searcher threads then begin executing the backtracking algorithm on their own event-data set.

When one of the searcher threads finds a match to the pattern, it acquires a "put" lock, inserts the match into a list, inserts its semaphore into a list of semaphores, wakes up the master thread, and then releases the "put" lock before going to sleep on its own semaphore. The awoken master thread then wakes up the main thread and goes to sleep until the user asks for another match. When the user asks for another match, the main thread wakes up the master thread which in turn wakes up the search thread whose semaphore is at the front of the list of semaphores. If other threads have returned more matches, the master thread informs the main thread about it as before, otherwise, it goes back to sleep. The use of a semaphore per task resulted in a slight improvement in performance relative to when one semaphore was used as the latter approach caused the searcher threads to block for longer periods of time when submitting a match.

When a searcher thread completes a task, it acquires a "get" lock in order to retrieve the next available task. Each time a task is completed, a counter is incremented. Using this counter, a thread is able to know when it has completed the last task (i.e., when the counter is equal to the number of tasks), at which point it wakes up the master thread which informs the main thread that there are no more matches.

Chapter 5

Experiments and Results

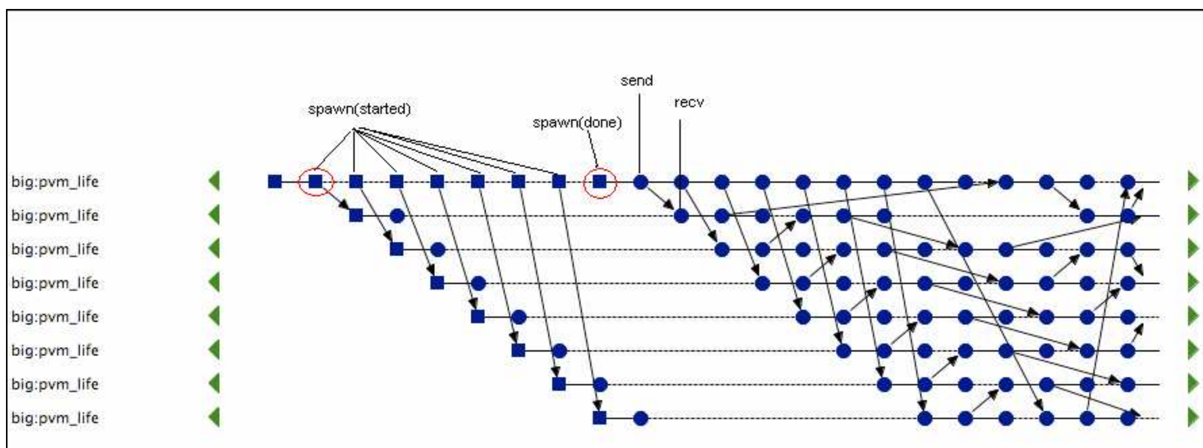
5.1. Test Setup

The parallel algorithm is implemented in Eclipse POET which was developed as an Eclipse plug-in using Eclipse 3.3 or later [1]. It also requires the Eclipse Graphical Editing Framework (GEF) 3.3 or later and a database that stores the event-data set. The databases currently supported are hsqldb [2], MySQL [3], and PostgreSQL [4]. Our experiments were performed using the MySQL database. Eclipse POET supports importing UEF data sets which are plain text files that contain the events from the target environment. Eclipse POET also provides a user interface for viewing the data sets and searching for patterns. To import a data set into the database, we start up a second instance of Eclipse by choosing "Run" → "Open Run Dialog ..." and then double-clicking on "Eclipse Application". Clicking on "Run" in the pop-up dialog will automatically include any plug-ins and source-code packages in the workspace.

To import a new partial-order data set, a new project must be created via "File" → "New Project". We can then import a new data set by right-clicking on the new project and selecting POET's event-database wizard. The wizard asks for the database to connect to, some database credentials, the UEF file, and the target-descriptor file. The target-descriptor file contains the information needed to map events from the target application

into POET events. Once all this information has been entered, the data set is imported into the database and we can view the data set on the screen.

To search for a pattern, we click on the "POET" → "FindPatterns" menu-item which brings up a dialog box that enables us to select the pattern file to be used. Figure 5.1 shows POET's view of the PVM Life partial-order data set (located at poet/model/data/life.8.19.ef.uef). The application that produced this data set consists of eight processes. The simulation starts out with a parent process spawning seven processes and then sending a message to all these processes. Each child process then selects a neighbour and forwards the message to it. The figure shows a "*spawn(started)*" event and then a "*spawn(done)*" event which signifies that all child processes were started and spawned successfully. The "*send*" and "*recv*" events represent the events for sending and receiving messages respectively. The circled patterns in the figure show the first result returned by searching for the "Spawn" pattern. This pattern checks for every "*spawn(started)*" event that happens-before a "*spawn(done)*" event.




```
SpawnStart := ["", "spawn(started)", ""];
SpawnDone := ["", "spawn(done)", ""];
Spawn := SpawnStart --> SpawnDone;
```

Figure 5.1: PVM Life Event Data Set

In order to conveniently run batch tests, we use command-line tools to import the event data sets and search for patterns instead of the user interface described above. The *poet.model.importer.UEFImporter* tool imports an event data set while the *poet.core.pattern.Searcher* tool is used for pattern matching. The parameters used to invoke these tools are shown in Figure 5.2. Note that we used re-written patterns in all the experiments.

```
UEFImporter: java poet.model.importer.UEFImporter <dbname> <username> -p<password> <UEF>
<target desc. file>

Searcher: java poet.core.pattern.Searcher <dbname> <pattern_file> <pattern_name> <mode> <opt>
        where <mode> is "flat" for re-written patterns OR "no_flat" for the original pattern format
        and <opt> is "true" to use the optimization for universal quantifiers or "false" otherwise
```

Figure 5.2: Command-Line Parameters for POET Tools

```
Preloading cache: 32 milliseconds (not included in total time)
================================================
Mode [ParallelOneMatch]
Pattern Init: 8 milliseconds (included in total time)
Number of threads - 2
Number of subtasks - 4, time - 6 milliseconds
1.    96 milliseconds    [[0,8], [0,1]]
2.    0 milliseconds    [[0,8], [0,3]]
3.    0 milliseconds    [[0,8], [0,2]]
4.    0 milliseconds    [[0,8], [0,4]]
5.    1 milliseconds    [[0,5], [0,8]]
6.    0 milliseconds    [[0,7], [0,8]]
7.    0 milliseconds    [[0,6], [0,8]]
Number of timestamps 7
Total 7 matches found in 107 milliseconds.
<5 seconds: 7
<10 seconds: 0
<30 seconds: 0
<60 seconds: 0
<300 seconds: 0
```

Figure 5.3: Search-Tool Example

Figure 5.3 shows the results from using the search tool when searching for the pattern in Figure 5.1 on the PVM Life data set. The tool displays one matched event per line, as a sequence of vector timestamps, as well as the time taken to find that match. The

tool also displays the total number of timestamps needed to find all the matches, and the total time taken. The last few lines show how many matches were found within 5, 10, 30, 60, or 300 seconds. Note that the tool finds all matches by simulating the user requesting the "next" match so the response time would be similar to the response time when using the user interface (assuming the user could click the "next" button very fast).

5.2. Performance Evaluation

In this section, we compare the performance of the sequential algorithm with the parallel version. We also evaluate the performance of the two task-creation strategies. Performance is measured based on how long it takes for the algorithm to return all the matches for a given pattern. For all the results shown, we deduct the time it takes to retrieve the events from the database. In these tests, we use four different data-sets and eight different patterns. Several of the patterns are taken from Nichols' thesis [32]; a few additional patterns and larger data sets were also used. We describe the patterns used in order to illustrate how pattern matching can guide a developer when diagnosing faults in a distributed system. Also, we use larger data sets in order to evaluate the performance of the search algorithm as many real-world applications could have up to 100,000 events.

The first two data sets are taken from the PVM environment. The first one is obtained from the distributed merge-sort application containing 16 traces and 138 events (binarymerge.16.29.ef.uef). The first pattern from this set is "ConSend8" which finds how many instances of eight concurrent "*send*" events there are in the data set. The results show that there are 32 matches. Given that there are 16 traces, it is plausible to still find eight concurrent sends with the other eight processes being on the receiving end of the message. This type of pattern gives us a sense of how much concurrency the application achieves and could help diagnose performance problems. The next pattern is "ConSend9" which finds out if there are nine concurrent "*send*" events during the program execution. We expect that this should be impossible and the results return 0 matches.

```
Send := ["", "send", ""];
Recv := ["", "recv", ""];
ConSend9 := (Send || Send || Send || Send || Send || Send || Send || Send || Send);
ConSend8 := (Send || Send || Send || Send || Send || Send || Send || Send);
SendRecv := (Send -(ANY) -> Recv);
SendSend := (Send -(ANY) -> Send);
```

Figure 5.4: PVM Patterns

The next data set is from the PVM life application whose operation was described in the previous section. The data set used here is much larger than the previous one, containing 128 traces and 31,098 events, and with a file size of 332 KB. The third pattern, "SendSend" which contains a universal quantifier when it is re-written searches for two consecutive "*send*" events. Here, we see the use of the limited operator excluding any event between the two "*send*" events. This pattern returns 127 matches, as expected, since the only time consecutive sends occur is at the beginning of the program when the parent process sends a message to all the other processes. Any other "*send*" event from a child process would be followed by a "*recv*" event as the child process must wait to receive a message before sending it out to a neighbour. The fourth pattern taken from this data set is "SendRecv" and it also contains a universal quantifier when it is re-written. This pattern simply counts how many successful data transfers occurred during the execution of the program. In this case there are 7678 matches.

The third data set used was collected from a $\mu C{+}{+}$ application containing an intentional bug. The bug sometimes allows a method that should be mutually exclusive to be accessed by more than one thread. The data set contains more than 177,735 events over eleven traces and is 4.3 MB in size. The fifth pattern is called "ConcurrentMonitors" and it checks the number of times more than one thread is present in the method. There are 65 occurrences, therefore the bug is discovered. The sixth pattern "StartStop", verifies that threads are started and stopped correctly by counting how many times a "thread start" precedes a "thread stop".

```
EnterMonitor1 := ["M1(0x0x9ac5730)", "thread received", ""];
EnterMonitor2 := ["M1(0x0x9ac5684)", "thread received", ""];
ConcurrentMonitors := EnterMonitor1 || EnterMonitor2;
ANY := ["", "", ""];
ThreadStart := ["", "thread start", ""];
ThreadStop := ["", "thread stop", ""];
StartStop := (ANY --> ThreadStart) --> ThreadStop;
```

Figure 5.5: µC++ Patterns

The final data set used was collected from the TCP-socket application as described in Section 2.4.1. This data set contains 240,561 events and is about 21MB in size. The “FirstConnectionEstablished” and “LastConnectionEstablished” patterns are the seventh and final patterns respectively (see Figure 2.8).

The experiments were run on a Linux Ubuntu SMP server with four 1.8-GHz Six-Core AMD Opteron processors, for a total of 24 cores, and 66 GB of RAM. L1, L2, and L3 cache sizes are 128 KB, 512 KB, and 6144 KB respectively. Each test was repeated five times and the results of all patterns (except Patterns 1 and 2) were not more than 5% from the average result. Some results from Patterns 1 and 2 differed by up to 10% and 25% respectively from the average. It is important to note that in both the sequential and parallel modes the variable re-ordering algorithm could produce different orderings of the same pattern that have very different execution times. This was seen in Pattern 6. For this reason, we selected only the test runs that had the same variable ordering.

The tables below show the average time taken for the existing sequential algorithm and the parallel algorithm using two and four cores with a sub-task factor of two, as well as the results of the two task-creation strategies.

Table 5.1: Execution Time for Sequential and Parallel Algorithms on 2 Cores

Pattern	Sequential	Grouped	Scattered	Grouped Speed-up	Scattered Speed-up
1	41.1	26.5	35.4	1.54	1.15
2	66.6	42.3	50.1	1.57	1.32
3	117.6	107.2	129.9	1.09	0.90
4	305.7	172.6	186.9	1.77	1.63
5	9.5	5.4	5.3	1.73	1.78
6	9.8	5.5	5.0	1.78	1.95
7	24.8	0.9	1.1	27.10	22.54
8	29.8	1.0	1.1	27.93	26.53

Table 5.2: Execution Time for Sequential and Parallel Algorithms on 4 Cores

Pattern	Sequential	Grouped	Scattered	Grouped Speed-up	Scattered Speed-up
1	41.1	20.8	23.3	1.97	1.83
2	66.6	20.6	37.0	3.22	1.79
3	117.6	61.0	75.5	1.92	1.55
4	305.7	106.2	108.0	2.87	2.83
5	9.5	3.4	3.4	2.78	2.78
6	9.8	2.8	2.9	3.41	3.33
7	24.8	0.6	0.6	37.12	37.12
8	29.8	0.6	0.6	44.46	44.46

5.2.1. Evaluation of Task-Generation Strategies and Optimizations

From the results above, it is seen that for Patterns 1 to 3, the *grouped* task generation method performs better than the *scattered* method by about a 20% decrease in execution time on average. For Pattern 2, the results on four cores show greater performance degradation when using the scattered method. This is because this approach results in a more imbalanced search tree where the first thread finishes within the first 15 seconds and so is left idle for the remaining 20 seconds. For Patterns 4 to 8, the two methods performed similarly. These results suggest that on average, the grouped method performs better than the scattered approach. This is most likely because maintaining the initial ordering of the primitive events as they occurred on their target applications is

more favourable to the pattern-matching algorithm than scattering this order in an attempt to achieve more balanced tasks. We also suspect that the grouped approach performs better because within each task, memory locations in close proximity are accessed within the same time period (i.e., it favours spatial locality).

For Pattern 3, which contains universal quantifiers, the results of the parallel algorithm are not very good especially when the *scattered* method is used. For the *grouped* method on two and four cores, we see a speed up of only about 1.09 and 1.92 respectively. For the *scattered* method on four cores, the speed-up achieved by the parallel algorithm is only 1.55 and on two cores the parallel algorithm is actually slower than the sequential one. These poor results occur because the optimization of the sequential algorithm for the universal quantifiers (see Section 4.4) outperforms the parallel algorithm. Using the optimized parallel algorithm for universal quantifiers discussed in Section 4.4, we repeated the tests for Patterns 3 and 4, and the results are shown in Tables 5.3 and 5.4.

Table 5.3: Execution Time of Non-Optimized vs. Optimized Algorithms on 2 Cores

Pattern	Grouped		Scattered		Grouped Speed-up		Scattered Speed-up	
	Non-Opt	Opt	Non-Opt	Opt	Non-Opt	Opt	Non-Opt	Opt
3	107.2	94.6	129.9	107.1	1.08	1.24	0.90	1.09
4	172.6	164.2	186.9	177.5	1.77	1.86	1.63	1.72

Table 5.4: Execution Time of Non-Optimized vs. Optimized Algorithms on 4 Cores

Pattern	Grouped		Scattered		Grouped Speed-up		Scattered Speed-up	
	Non-Opt	Opt	Non-Opt	Opt	Non-Opt	Opt	Non-Opt	Opt
3	61.0	59.0	75.5	62.8	1.92	1.99	1.55	1.87
4	106.2	90.5	108.0	95.2	2.87	3.37	2.83	3.21

The results for Pattern 3 (see Figures 5.6 to 5.7), show that the optimized parallel algorithm is about 12% faster than the non-optimized version when using the grouped method. On four cores, the grouped method of the optimized and non-optimized versions perform similarly for this pattern. Using the scattered method on both two and four cores, the optimized parallel algorithm is about 15% faster than the non-optimized version. Note

that each point on the graph is the average speed-up while the top and bottom points of each bar represent the maximum and minimum speed-up (respectively) of the five runs.

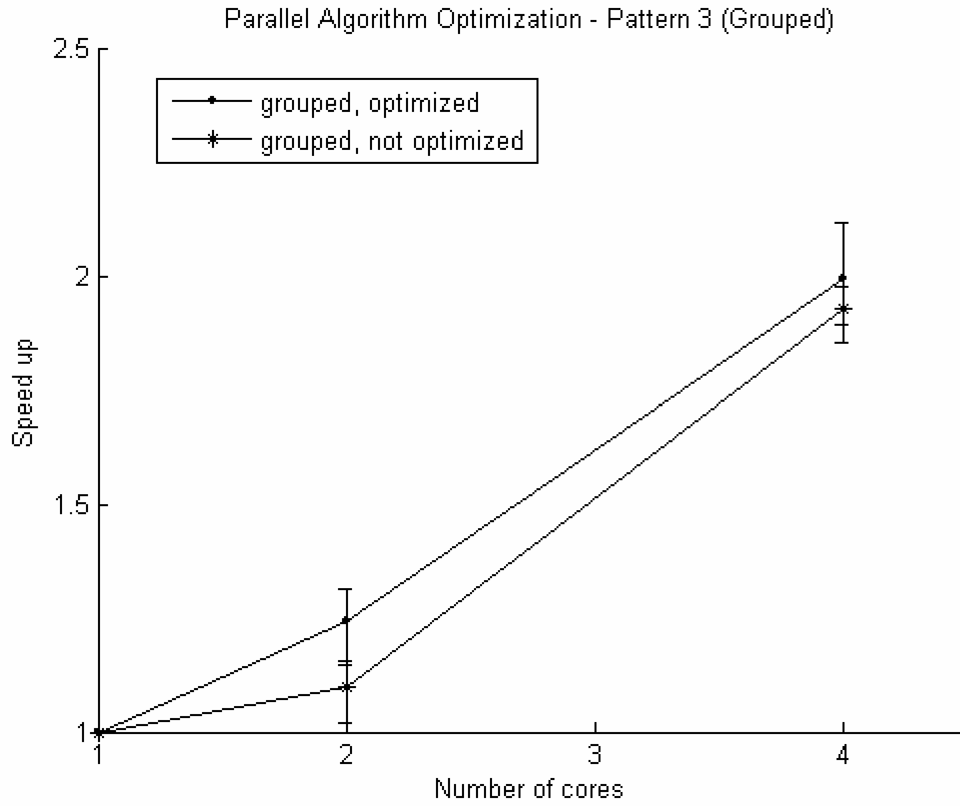


Figure 5.6: Optimized Parallel Algorithm for Universal Quantifiers (Pattern 3 - Grouped)

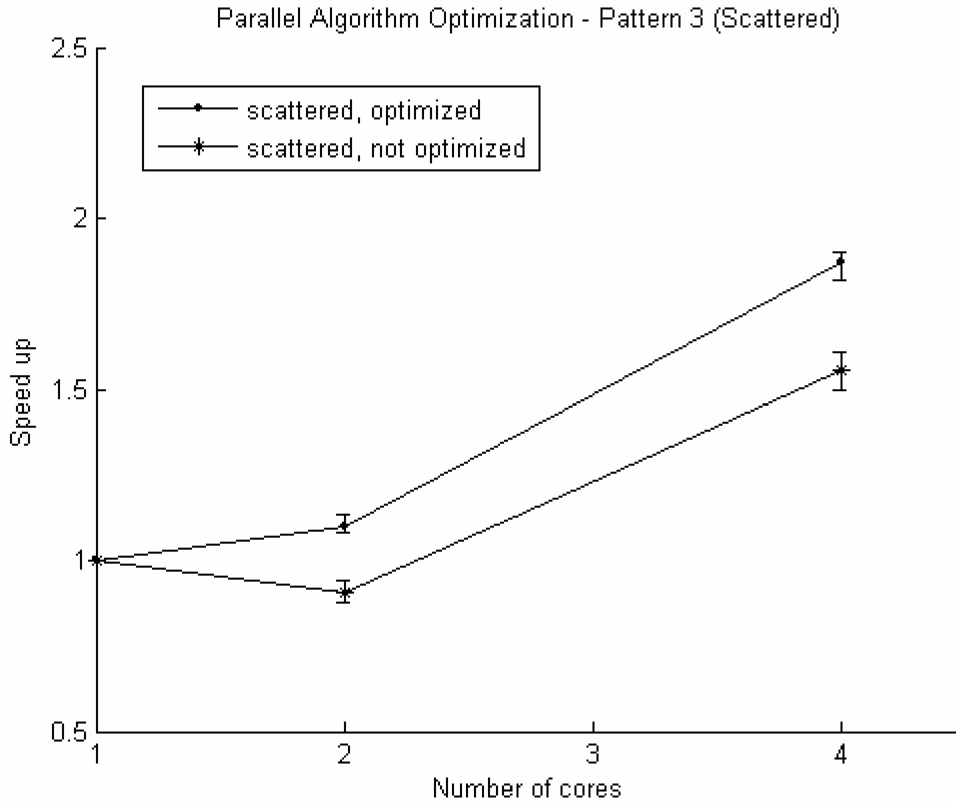


Figure 5.7: Optimized Parallel Algorithm for Universal Quantifiers (Pattern 3 - Scattered)

For Pattern 4 (see Figures 5.8 to 5.9), when using the grouped and scattered methods on four cores, we see that the optimized parallel algorithm is about 15% and 12% faster respectively than the non-optimized version. There is not much performance difference between the optimized and non-optimized versions of both methods when this pattern is run on two cores. The graphs in Figures 5.6 to 5.9 show the speed-up achieved when using the optimized parallel algorithm on these patterns and we see that in certain cases the optimization does improve performance.

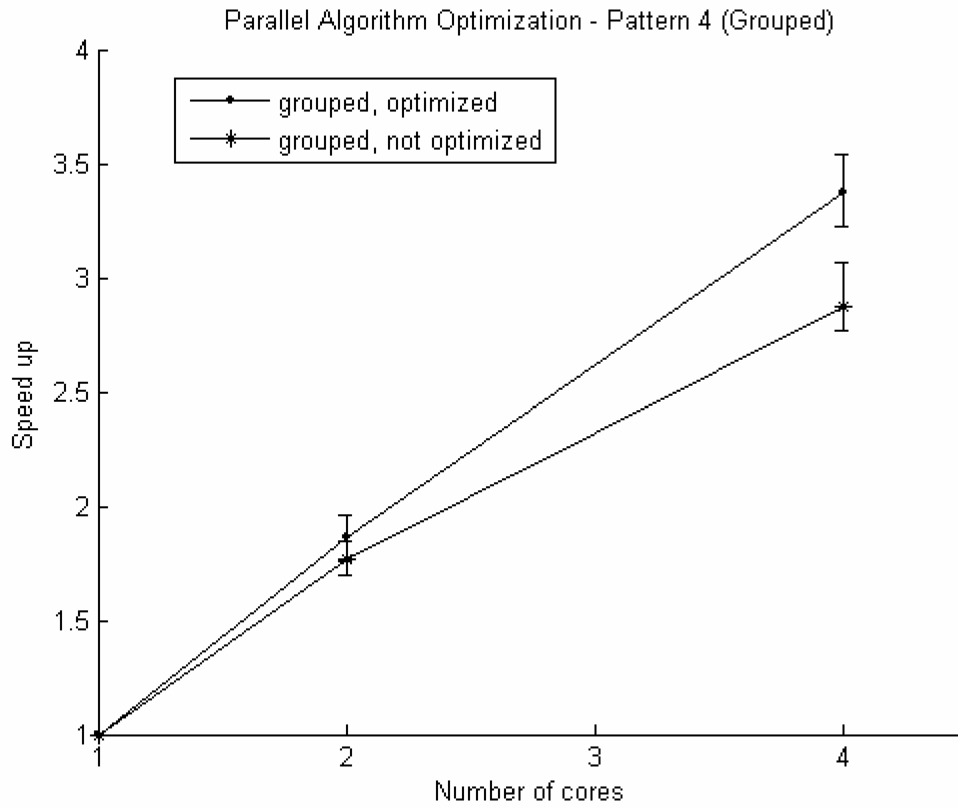


Figure 5.8: Optimized Parallel Algorithm for Universal Quantifiers (Pattern 4 - Grouped)

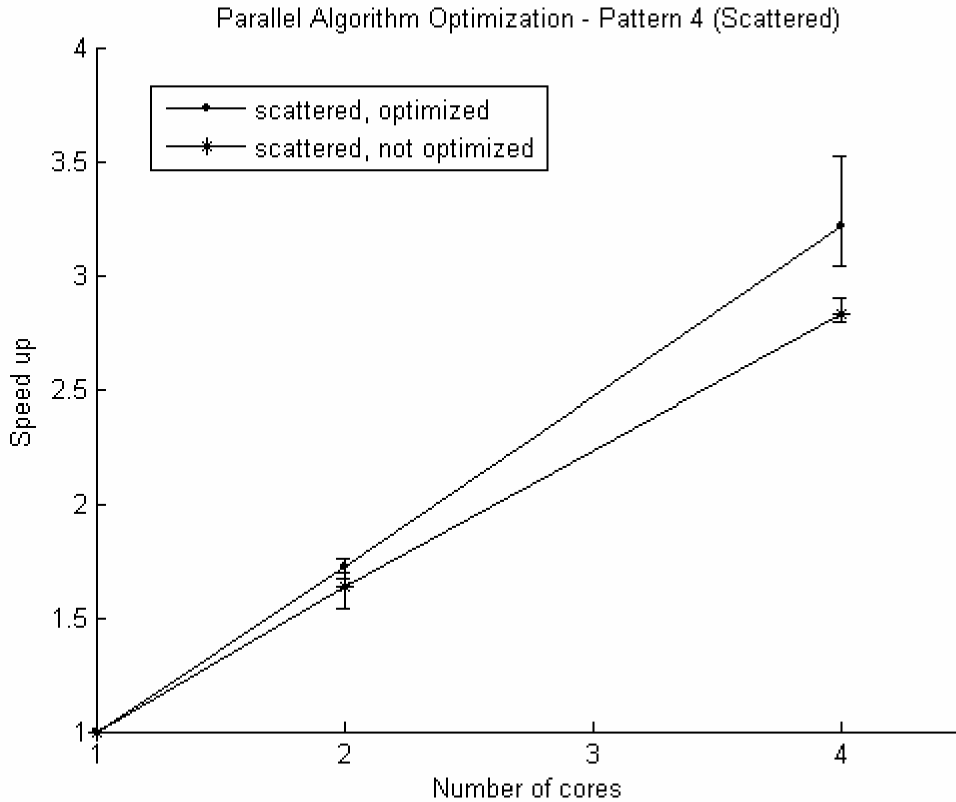


Figure 5.9: Optimized Parallel Algorithm for Universal Quantifiers (Pattern 4 - Scattered)

5.2.2. Parallel-Algorithm Evaluation

In this section, we evaluate the speed-up of the parallel algorithm by considering only the grouped method. We focus on the results on four cores referring back to Table 5.2 for Patterns 1, 2, and 5 to 8, and Table 5.4 for Patterns 3 and 4. For the first pattern, we see that the speed-up achieved is only 1.97. The second pattern, "ConSend9" which is similar to "ConSend8", has a much better speed-up of 3.22. It is useful to point out that one of the five runs for this pattern had a speed-up of 2.40 (i.e., about 25% worse). It should be noted here that of all the patterns tested, this was the only pattern that showed this much disparity in execution time. A closer look at the "ConSend9" pattern in its re-written form shows that though there are 9 distinct variables, each variable refers to the same domain space i.e., *send* primitive events. "ConSend8" has similar properties. This

leads us to suspect that the reason for the poor performance in “ConSend8” and much disparity in runtimes in “ConSend9” may be attributed to memory allocation and access patterns that influence cache hit or miss rates.

For Pattern 3 (“SendSend”), a speed-up of 1.99 is not very encouraging and it shows that for this pattern, the optimization for universal quantifiers is not sufficient to achieve a much better speed-up. On the other hand, the next pattern, “SendRecv”, has a speed-up of 3.37 indicating that for this pattern, the optimization of universal quantifiers does a much better job of improving the speed-up.

For Patterns 5 and 6, a speed-up of 2.78 and 3.41 is achieved which is quite good considering that these patterns only run for about three seconds on four cores.

For Patterns 7 and 8 we see that the result from the parallel algorithm is about 40 times as fast as the sequential algorithm. The reason for the tremendous speed up peculiar to these patterns is some inefficiency in the sequential algorithm that is avoided in the parallel algorithm. When Pattern 7 is re-written (see Appendix B), it consists of fourteen disjunctions with the first eleven disjunctions each consisting of eight happens-before pairs. The second to the eighth set of disjunctions are exactly the same except for the last happens-before pair. Similarly, the ninth to twelfth disjunctions are exactly the same except for the last happens-before pair. The similarity among the disjunctions contributes significantly to the tremendous speed-up achieved.

In finding a match for this pattern, the backtracking algorithm starts out as usual. After finding the only match, the algorithm backtracks to the ninth disjunction and picks the next value of the variable that was last assigned. This variable is the “DoneConnect” variable and it is assigned to its next value. At this step, the new value assigned to this variable is not bound during the evaluation of the disjunction because a happens-before pair that contains other variables satisfies the disjunction. These other variables have the same values that had been found in the first match and so the resulting match that occurs due to this step is a duplicate result. Note that because of the similarity of the disjunctions, a similar scenario is repeated at each disjunction resulting in a lot of unnecessary work. On the other hand, the parallel algorithm is able to avoid this work because it selects the “DoneConnect” variable as one of the *fixed* variables; as such, its domain is split into a set of one (as the initial size of the domain is two). Therefore, after

the first match is found, there are no more values to be assigned to this variable and so all the unnecessary work that led to duplicate results in the sequential algorithm is avoided. One would expect that it would be difficult to predict when splitting the search tree in order to achieve parallelism would result in this kind of performance boost; however, it may be possible to optimize the sequential algorithm by looking at the properties of the re-written pattern and eliminating unnecessary work while the algorithm is running.

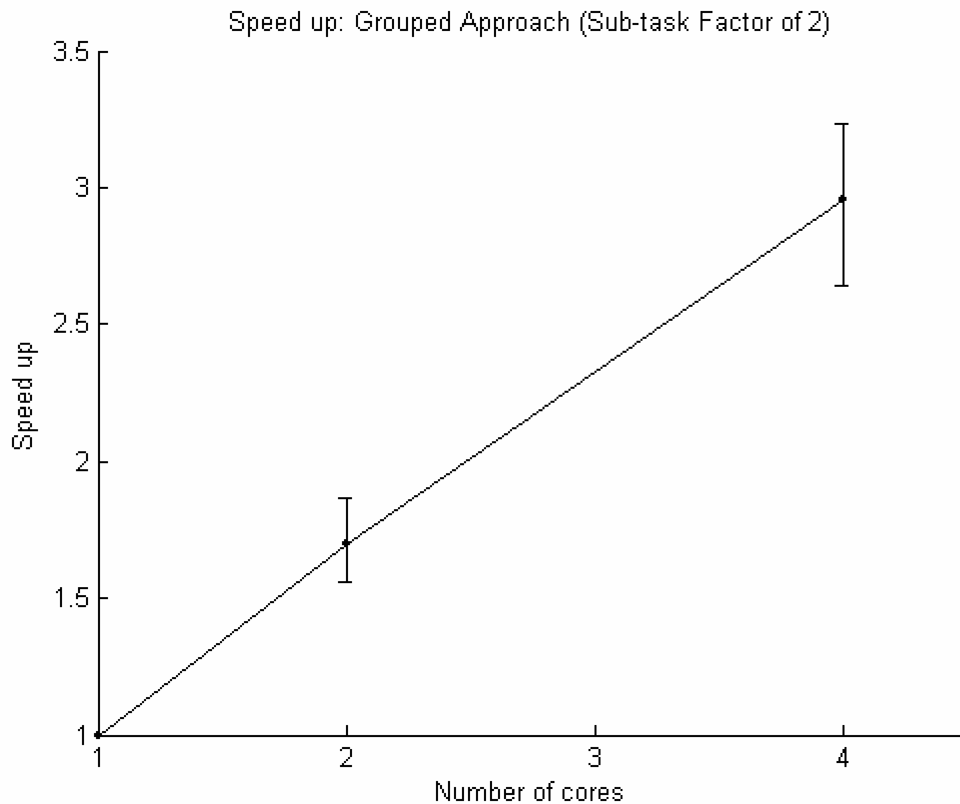


Figure 5.10: Speed-up Grouped Approach (Sub-task Factor of 2)

The graph in Figure 5.10 shows the speed-up obtained as the number of cores increases when using the grouped task-generation methods with a sub-task factor of 2. Note that the graph shows the results for Patterns 1, 2, and 4 to 6. In order to clearly visualize the average performance, we excluded Pattern 3 where the optimization of universal quantifiers did not help much, and Patterns 7 and 8 where the parallel algorithm outperformed the sequential one by over a factor of 30. From the graph, we see that the average speed-up for the grouped method on two cores and four cores is about 1.70 and

3.0 which is equivalent to an efficiency of 85% and 75% respectively. Note that the efficiency refers to the utilization of the cores and is given by the speed-up divided by the number of cores.

5.2.3. Task-Size Analysis

Next we consider how the number of tasks generated affects performance. We ran the parallel algorithm using 4, 8, 16 and 32 tasks on four cores (i.e., a sub-task factor of 1, 2, 4 and 8, respectively), and found that for all the patterns, it took less than 1 second to create the tasks. Table 5.5 shows the average time taken to run the parallel algorithm on four cores using the *grouped* method. Note that there are no results for 16 and 32 tasks for Patterns 7 and 8 as the search tree could not be split into more tasks. From the table, it is seen that using four tasks performs 40% worse (on average) for Patterns 1 and 2 than when 8 tasks were used. This is the most noticeable difference between 4 and 8 tasks among all the patterns. The reason for this poorer performance when using 4 tasks is that the task distribution of Patterns 1 and 2 is very unbalanced with the first thread finishing in under a second and then becoming idle for the remainder of the execution time.

Table 5.5: Total time of the Parallel Algorithm

Pattern	4 Tasks	8 Tasks	16 Tasks	32 Tasks
1	30.7	20.8	22.0	20.0
2	38.4	20.6	19.0	18.9
3	57.1	59.0	48.5	49.1
4	88.7	90.5	92.6	83.7
5	3.5	3.4	3.3	3.3
6	2.8	2.8	2.9	2.9
7	0.8	0.6	-	-
8	0.9	0.6	-	-

For Pattern 3, we see that the execution time between 4 and 8 tasks is similar whereas using 16 tasks performs 17% better than when 8 tasks are used. Again, this is because the threads are idle for a shorter time period when more tasks are used. For all the other patterns, the execution time between the various tasks sizes differed by not

more than 10% from each other. In summary, Figure 5.11 shows that as the number of tasks increases, the speed-up achieved remains at relatively the same level. Therefore, we suggest that using a sub-task factor of 8 may be sufficient for most patterns in order to avoid thread starvation.

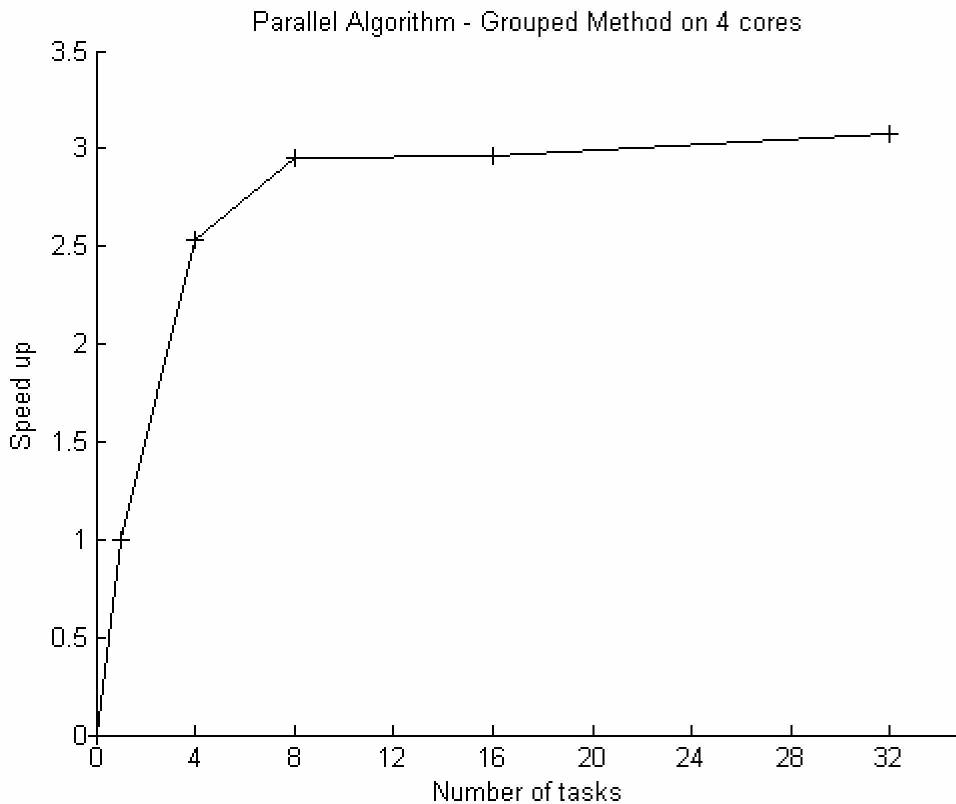


Figure 5.11: Speed-up Using Various Sizes of Tasks

5.3. Summary

Based on the results from the performance experiments, we can make certain recommendations on how to configure the parameters in order to achieve optimum performance of the parallel algorithm. Because the search algorithm is CPU-intensive with no I/O-bound operations, the number of threads should be equal to the number of cores on the computer. Obviously, using fewer threads will not fully utilize all the cores

and initial experiments using more threads showed no performance improvement. The number of tasks generated should be about eight times the number of cores as generating fewer tasks creates a higher chance of having idle threads for longer periods of time. On the other hand, it is not recommended to have a task size of more than eight times the number of cores, as this is not expected to produce a significant performance improvement. Finally, the experiments have shown that for most patterns, using the grouped approach for task-generation performs better than the scattered approach.

To conclude, the graphs shown in Figures 5.6 to 5.11 reveal that the parallel algorithm does not achieve linear speed-up. This is probably unattainable for many parallel algorithms; however, attaining an efficiency of up to 75% (on average) on four cores is quite good. In the next chapter, we take a look at an approach to dynamic work-stealing in the pattern-search algorithm and also investigate the scalability of the parallel algorithm as the number of cores increases.

Chapter 6

More Improvements and Experiments

In the previous chapter, we showed that generating a set of tasks that is eight times the number of processors (i.e., a sub-task factor of 8) is recommended as there was no significant improvement as the number of tasks increased. In addition, for all the patterns analyzed, using a size of tasks that is eight times the number of processors was sufficient in keeping all the threads busy throughout most of the algorithm's execution. In this chapter, we discuss a hybrid implementation of dynamic work-stealing in the pattern-search algorithm in order to further improve its performance. We illustrate the usefulness of this technique by introducing some patterns that reveal that even when a subtask-factor of eight is used, thread starvation is still possible.

6.1. Dynamic Work-stealing Algorithm

As previously mentioned, one of the challenges of any dynamic work-stealing strategy is to ensure that the cost of moving work from busy to idle threads is very small. To avoid this cost initially, our algorithm begins with a static approach with the size of tasks being eight times the number of processors. We further divide this set of tasks into $|T|/t$ groups, where t is the number of threads, and assign each group to a thread. As such, each thread has a local queue of tasks and when this queue is empty, it checks its neighbours for an available task. Stealing from neighbours first has the advantage of ensuring spatial locality thereby improving cache performance. If an idle thread finds a task from a

neighbour, it marks itself as SLOW_BY_STEALING and marks the neighbour it stole from as SLOW. If no task is found, the thread decrements a *finished* counter (initially set to the number of threads), and goes to sleep.

In this algorithm, the idle thread is initially responsible for looking for more work, but as tasks get depleted, the busy threads become responsible for splitting their work and giving it to idle threads. This is done mainly for efficiency of the algorithm as allowing idle threads to get work from busy threads would require a lot of synchronization effort. Also, experiments showed that the total cost polling to determine when to split a task and the cost of task-splitting itself by busy threads is insignificant compared to the execution time of the algorithm. Therefore, placing this extra burden on busy threads does not cause a significant performance overhead.

A busy thread periodically determines whether to split its work by checking if it is marked SLOW or SLOW_BY_STEALING, or if there are any idle threads (i.e., if the *finished* counter is less than the number of threads). If either of these conditions holds and the number of available tasks is less than the number of threads, the busy thread would split its work into two tasks if possible. The thread would put one task in a global queue, and then wake up an idle thread. The idle thread simply picks the work from the global queue and continues working. We suggest that the algorithm is efficient for the following reasons:

a) Initially, only busy threads that are marked SLOW or SLOW_BY_STEALING split their work. This ensures that the portions of the tree that are "difficult" are split into smaller tasks and given to other threads. Without this condition, threads could end up splitting tasks that do not take long to complete which could cause unnecessary overhead.
b) The previous condition in (a) is relaxed when there are idle threads, at which time any busy thread can split its work. This ensures that threads are not idle for too long.
c) Finally, tasks are split only when the total number of available tasks is less than the number of threads. This ensures that busy threads do not start splitting work unnecessarily when there is still enough available. It also ensures that the global

queue starts getting filled as other less busy threads complete their work thereby preventing threads from being idle for too long.

The algorithm determines that there is no more work when the *finished* counter reaches zero. The thread that decrements the counter to zero would notify the master thread and wake up all sleeping threads which then terminate.

6.1.1. Task Splitting

As previously mentioned, a busy thread periodically checks whether it should split its task. This is done using a counter configured by the *poet.core.searcher.parallel.peektime* parameter in the #*poet.properties* file. This counter is equivalent to the number of nodes a thread should visit when executing the backtracking algorithm before checking to see if the conditions for task-splitting hold. This parameter is perturbed a little for each thread to further ensure that task-splitting occurs at different times among the threads.

Task-splitting occurs closest to the root of the search tree in order to ensure that the work is large enough. The algorithm begins with the first level of the tree and splits the unvisited set of nodes into two (assuming that this level does not correspond to a universally quantified variable). If this is not possible because the thread is currently on the last node at this level, the algorithm moves down to the next level and tries to split the unvisited nodes at this level. Figure 6.1 shows task-splitting at various levels when the initial search tree is divided into two tasks that are handled by two threads.

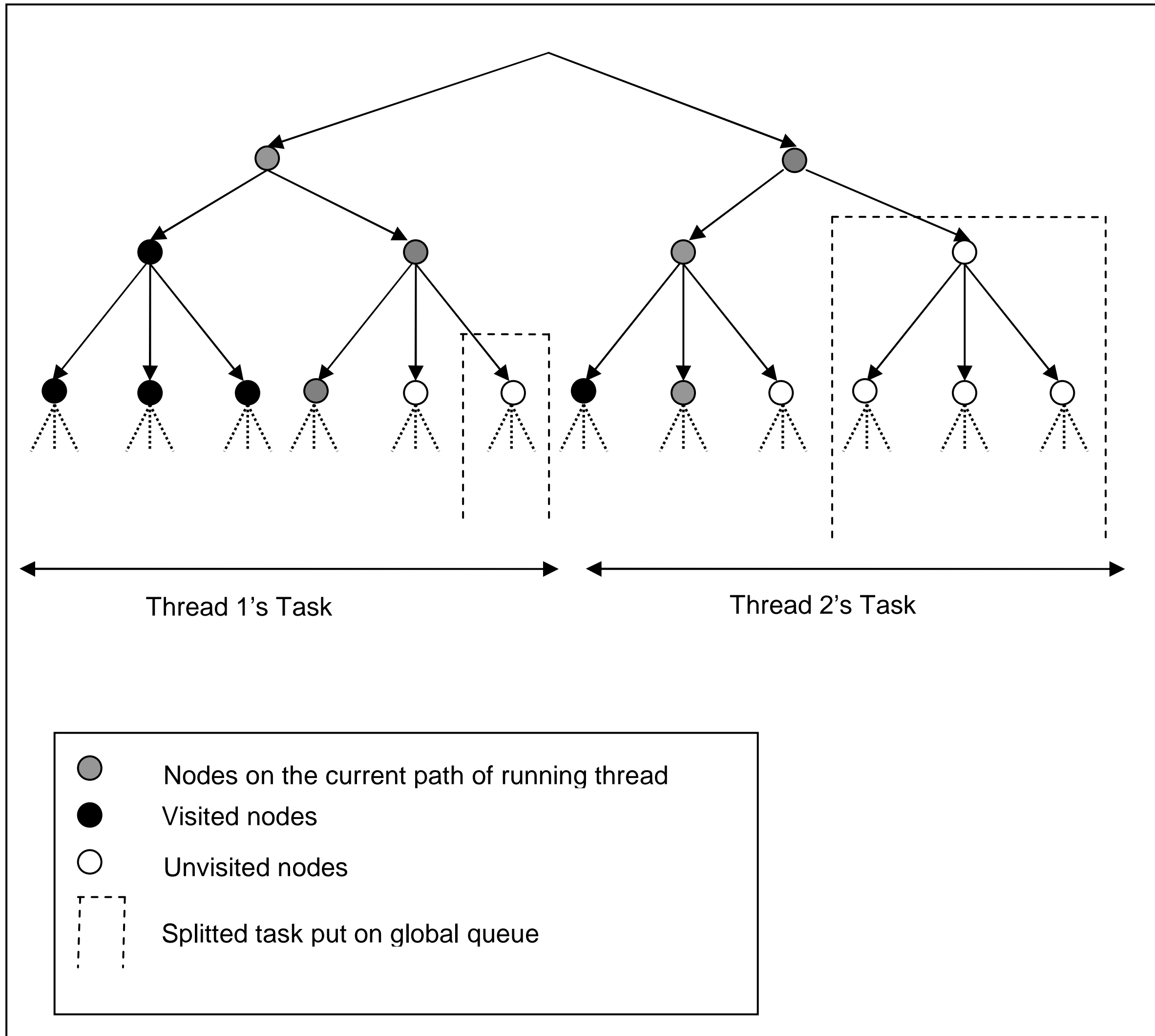


Figure 6.1: Task-splitting

6.2. Performance Evaluation

We evaluate the performance of the dynamic work-stealing approach by comparing it with the static approach using a subtask factor of eight. We suggest that an efficient dynamic work-stealing algorithm is one that performs as well as the static approach in cases where thread starvation does not occur and one that performs better otherwise. In addition to our existing patterns, we introduce a ninth pattern called "FinalDataTransfer" operating on the TCP-socket application dataset. This pattern finds the last data transfer that occurred from one of the clients to the server and is shown in Figure 6.2.

```
DataTransferC1 := ["Process9771", "Send", ""].["Closed44194", "Send_stream", ""];
DataTransferC2 := ["Process9777", "Send", ""].["Closed44200", "Send_stream", ""];

DataTransferC1 *alldtc1;
DataTransferC2 *alldtc2, $dtc2;

FinalDataTransfer := ($dtc2 !--> *alldtc1) & ($dtc2 !--> *alldtc2);
```

Figure 6.2: "FinalDataTransfer" Pattern

We also include a fifth dataset taken from the random-communication application. This MPI [4] application generates communication events with no specific regularities. The dataset contains 53,248 events, consists of 251 processes and is 1.5 MB in size. Each process in this application repeatedly selects at random another process to send a message to. The first pattern from this dataset (the tenth overall) is called "FourSendSendP1" and finds the instances of four consecutive send events that occur on the first process. There are only five matches returned out of 100 send events on this process. The second pattern, "TwoSendRecvP1", finds instances of two consecutive send and receive pairs that occur on the first process. There are 17 matches found of a total of 201 send and receive events. The final pattern from this data set, "ConSendP1P9" checks for two consecutive receive events on the ninth process that could potentially have come from two consecutive send events on the first process. The pattern "SendSendP1" checks for two consecutive sends on Process 1 while "RecvRecvP9" checks for two consecutive receive events on Process 9. "ConSendP1P9" then consists of these two patterns. There are total of 552 matches returned for this pattern. All these patterns aim at verifying that the communication application is indeed random with no form of regularities. The patterns are shown in Figure 6.3.

```
ANY := ["", "", ""];
P1Send := ["Process 1", "send", ""];
P1Recv := ["Process 1", "receive", ""];
P9Recv := ["Process 9", "receive", ""];

P1Send $p1, $p2, $p3, $p4;
P1Recv $r1, $r2;
FourSendSendP1 := ($p1 -(ANY)-> $p2) & ($p2 -(ANY)-> $p3) & ($p3 -(ANY)-> $p4) ;
TwoSendRecvP1 := ($p1 -(ANY)-> $r1) & ($r1 -(ANY)-> $p2) & ($p2 -(ANY)-> $r2);

SendSendP1 := (P1Send -(ANY)-> P1Send);
RecvRecvP9 := (P9Recv -(ANY)-> P9Recv);
ConSendP1P9 := SendSendP1 --> RecvRecvP9;
```

Figure 6.3: Random Patterns

The table below shows the execution time of the static task-distribution method using a sub-task factor of eight (i.e., 32 tasks, except patterns 7 and 8 which use 8 tasks) versus the dynamic approach. For the dynamic approach, we use a peek-time of 100. The tests were executed on four cores and each test was run five times except for Patterns 9 and 12 that were run three times due to their long execution times. For Patterns 3 to 11 and Pattern 12 each run time did not differ by more than 5% and 7% from the average respectively. Some results from Patterns 1 and 2 differed by up to 10% and 25% respectively from the average.

We see that for Patterns 1 to 4, and 6 to 9 the results from both strategies differ by less than 10%. So we can conclude that the two strategies perform similarly for these patterns. For Pattern 5, the static strategy performs about 15% better than the dynamic approach. Since the execution time of this pattern is less than 4 seconds, the overhead of the dynamic approach outweighs performance improvement. For Patterns 10 to 12, the execution time of the dynamic approach is 15% to 25% faster than the static method. And we see that the running times for these are between 30 and 1100 seconds. Therefore, we can summarize that the dynamic approach performs reasonably well compared to the static approach without introducing too much overhead, provided the execution time is not very short (i.e., a few seconds).

Table 6.1: Execution Time for Static vs. Dynamic Strategies on 4 Cores

Pattern	Sequential	Static	Dynamic	Static Speed-up	Dynamic Speed-up
1	41.1	20.0	19.8	2.05	2.06
2	66.6	18.9	18.9	3.51	3.51
3	117.6	49.1	46.2	2.39	2.54
4	305.7	83.7	82.3	3.65	3.71
5	9.5	3.3	3.8	2.83	2.44
6	9.8	2.9	3.2	3.30	2.98
7	24.8	0.6	0.7	37.12	35.14
8	29.8	0.6	0.7	44.46	42.11
9	2150	569.8	567.9	3.77	3.78
10	103.6	39.4	29.5	2.62	3.50
11	221.5	77.2	61.9	2.86	3.57
12	3994.4	1296.4	1082.5	3.08	3.69

6.3. Scalability of the Parallel Algorithm

In this section we evaluate the level of parallelism we achieve as the number of cores is increased. Although POET would generally not be run on large servers with hundreds of processors, we include this section in order to measure the scalability of the algorithm to a modest number of cores. We select patterns that run for several minutes in order to justify the need for more cores. We select Pattern 5, SendRecv, which takes 5 minutes to run in the sequential mode. We also include Patterns 9 and 12 ("FinalDataTransfer" and "ConSendP1P9" patterns respectively) that run for 35 and 65 minutes in the sequential mode. We compare the scalability of the static versus dynamic task-distribution strategies in the results in Tables 6.2 and 6.3.

The tables show the time in seconds and speed-up of these patterns using the parallel algorithm with both the static and dynamic approach to task distribution. We use a sub-task factor of eight and a peek-time of 100. Patterns 4 and 9 were run five and three times respectively and the result from each run did not differ by more than 5% of the average. Pattern 12 was run three times and the result from each run did not differ by more than 10% from the average. From the table, we see that for both the dynamic and static strategies, the "SendRecv" pattern (Pattern 4) is about 80% efficient up to 8 cores,

and then drops to 50% efficiency at 16 cores and is only about 40% efficient at 24 cores. This is most likely because for a pattern that runs for 30 seconds on 16 cores, adding more cores means that there is probably not much work to keep them busy. On the other hand, for both approaches, the "FinalDataTransfer" (Pattern 9) and the "ConSendP1P9" (Pattern 12) patterns, which take a much longer time to run, are still up to 80% efficient at 24 cores. This suggests that the parallel algorithm is scalable for patterns that run for longer time periods.

Table 6.2: Execution Time of Parallel Algorithm on Several Cores

Pattern	Strategy	Number of cores							
		1	2	4	8	12	16	20	24
4	Static	305.7	158.6	83.7	47.5	36.0	34.5	34.7	31.7
4	Dynamic	305.7	155.6	82.3	47.3	37.7	30.2	27.4	25.7
9	Static	2150	1012.7	569.8	282.2	198.5	160.3	135.2	110.7
9	Dynamic	2150	1002.9	567.9	264.7	192.8	148.8	128.4	115.6
12	Static	3994.4	2480.6	1296.4	682.3	358.7	277.6	245.8	208.8
12	Dynamic	3994.4	2176.8	1082.5	505.9	377.9	290.4	229.4	201.6

Table 6.3: Speed-up of Parallel Algorithm on Several Cores

Pattern	Strategy	Number of Cores							
		1	2	4	8	12	16	20	24
4	Static	1	1.92	3.65	6.42	8.47	8.83	8.80	9.64
4	Dynamic	1	1.96	3.71	6.45	8.09	10.10	11.15	11.88
9	Static	1	2.12	3.77	7.61	10.82	13.41	15.89	19.41
9	Dynamic	1	2.14	3.78	8.12	11.15	14.44	16.73	18.58
12	Static	1	1.61	3.08	5.85	11.13	14.38	16.24	19.12
12	Dynamic	1	1.83	3.69	7.89	10.56	13.75	17.40	19.80

The tables above and the following graphs compare the speed-up of the dynamic versus static work distribution strategies. In Figure 6.4, we see that for the "SendRecv" pattern, the dynamic approach shows a significant improvement in speed-up relative to the static approach beyond 12 cores.

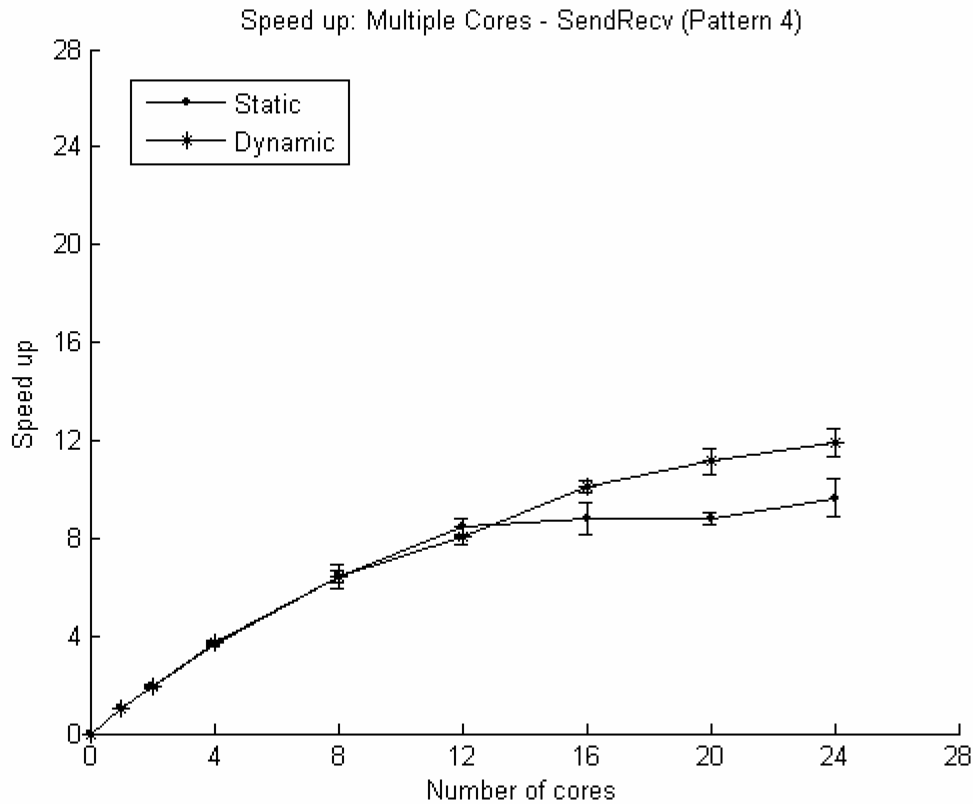


Figure 6.4: SendRecv - Speed-up on Multiple Cores

In Figure 6.5, the dynamic approach appears to show a slightly better speed-up on 16 and 20 cores (about 6% on average). On 24 cores, however, the static approach performs better by about 5%. In this case, we notice that there was no need for work-stealing as by time certain threads became idle, it was not possible for the busy threads to split their task any further. As such the dynamic work-stealing algorithm resulted in a slight performance overhead in this case.

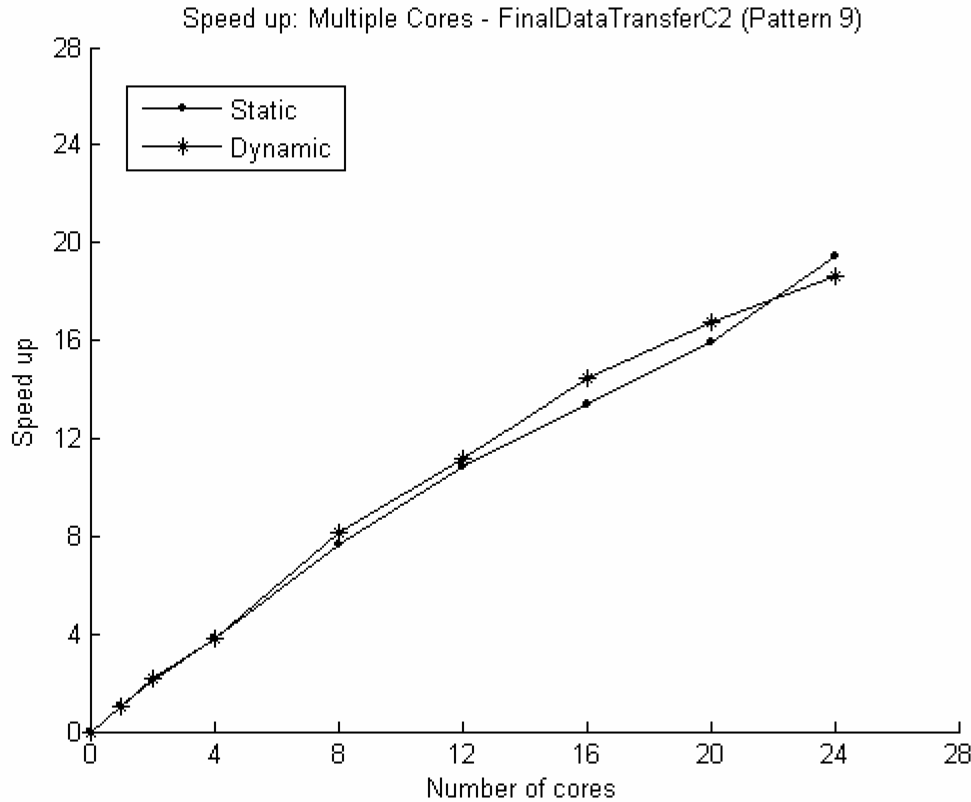


Figure 6.5: FinalDataTransfer - Speed-up on Multiple Cores[1]

Finally, in Figure 6.6, the dynamic approach performs significantly better on four and eight cores (15% and 25% better respectively) as it reduces any thread-starvation that occurs with the static approach. On 12 cores, we notice that the static approach is about 5% better on average because one of the runs finished in 332 seconds (i.e., 10% faster than the other two runs), and hence this resulted in a much better average performance than the dynamic approach. On 16 cores, the static approach is 5% better than the dynamic approach as the work-effort involved is more evenly distributed among the threads. Furthermore, we find that the dynamic approach in this case results in slight performance overhead where some busy threads were unable to split their work further and there was more contention for the few remaining tasks in the cases where tasks could be split. On 20 and 24 cores, the dynamic approach is 5% better as there is now a noticeable benefit to task-splitting. This suggests that even with more tasks (which is the

[1] In this graph, we omit the bars at each point as the two approaches have similar performance.

case as the number of cores increases), it is still possible to have a work-effort imbalance among the threads making dynamic work-stealing more favourable in such situations.

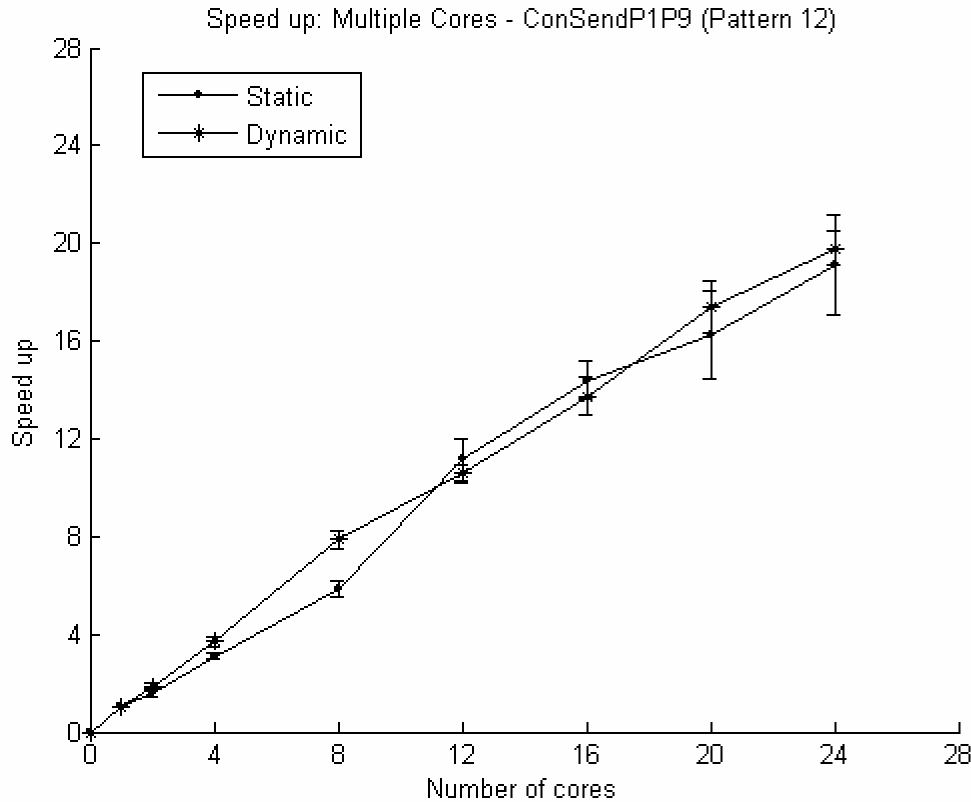


Figure 6.6: ConSendP1P9 - Speed-up on Multiple Cores

In summary, the dynamic approach does better than the static approach which is expected as threads are not left idle for long time periods. We also see that the dynamic approach scales equally well. One would expect that as the number of cores increases, thread starvation would be less likely as the search-tree would be divided into more tasks of smaller sizes. We see, however, that for the "SendRecv" pattern, the dynamic approach still results in a better performance improvement even beyond 12 cores which further highlights the benefits of this approach even when using several cores.

Chapter 7

Closing Remarks

7.1. Conclusions

Analyzing large event datasets emitted from distributed systems continues to be an area of research as proper analysis can enable developers to diagnose and fix faults faster. We showed how POET achieves this goal by providing a search algorithm that enables users to find event-patterns in large datasets. In this thesis, we developed an efficient and scalable parallel algorithm that improves the search process, making analysis of these datasets faster.

We have introduced techniques for distributing the search tree associated with the pattern-matching problem into several smaller tasks that can be independently handled by several cores. We have proved that the set of tasks is disjoint ensuring that cores are not repeating the same work-effort. We performed experiments with the grouped and scattered approaches of task-generation and showed that the grouped approach is more suited for the pattern-search problem in POET. This is because it maintains the ordering of the primitive events as they occurred in their target applications and it also improves the computer's hardware-cache performance due to ensuring spatial locality during memory accesses.

Patterns containing universal quantifiers are more challenging as they require comparing one event with all the events in an event class. Though the optimization for

universal quantifiers for the sequential algorithm is inherently difficult to parallelize, we introduced a simple optimization to the parallel algorithm that achieved up to 15% performance improvement over the generic parallel algorithm. We also showed that the static approach to task generation is not always sufficient even when we start out initially with a large number of tasks. As such, we introduced an efficient dynamic work-stealing algorithm that prevents processors from starving. Finally, our experimental results show that the parallel algorithm is scalable, providing efficiencies of up to 80% on 24 cores.

7.2. Future Work

The following are areas of further research that would improve the pattern-search feature of POET.

7.2.1. Improvements to Variable Re-ordering

The variable re-ordering algorithm in POET has been shown to improve the performance of the pattern-matching algorithm. However, the current heuristic used to determine the final ordering of a pattern needs to be improved. This is because the algorithm sometimes produces an ordering that performs significantly faster than another ordering it returned. Specifically, more work needs to be done to improve the method for breaking ties when selecting which disjunction should be chosen next in the final ordering. In the current scheme, ties are broken by choosing the disjunction with the fewest unassigned variable and then selecting the unassigned variables that occurs most frequently in the pattern. The problem is that there could be more than one disjunction that satisfies this condition. The current scheme simply picks the disjunction that is found first which could lead to different orderings (during different executions on the same pattern) that have varying execution times. A possible improvement may be to introduce a method that determines the probability of a variable being assigned. This may be useful for disjunctions with more than one happens-before pair where only the variables in one pair will actually be assigned when the search algorithm runs. Another possible improvement may be to break

ties by choosing disjunctions with fewer happens-before pairs as this indicates that such disjunctions may be satisfied with a smaller amount of work than those with more happens-before pairs.

7.2.2. Improvements to the Backtracking Algorithm

POET still uses a naïve backtracking algorithm which tends to go through a lot of steps repeatedly as demonstrated by an example in Section 4.1. A simple improvement that could be applied is the back-jumping technique described in Section 3.1.

7.2.3. Improvements Based on Re-written Patterns

The verbosity of re-written patterns can be used to improve the search algorithm as it reveals to some extent the steps the backtracking algorithm will follow. During some of our experiments we found that the search algorithm would find the same result as many as one hundred thousand times! This was seen with the "ConSend8", "FirstConnectionEstablished" and "LastConnectionEstablished" patterns. We see that the rules used to transform a pattern into its re-written form usually results in a lot of repetitions in constraints in the re-written pattern. We believe that initial analysis of the re-written pattern can be used to guide the search algorithm in order to avoid the unnecessary work of finding duplicate results that are later discarded.

7.2.4. Lower and Upper Bounds of Tasks

In Chapter 4, we presented the algorithm for task generation that avoids duplicate work-effort and generates a number of tasks approximately equal to the desired number *S*. It may be useful for the algorithm to have certain guarantees as to how "close" the actual number of tasks generated is to the initial desired number. In other words, can the algorithm guarantee that the number of tasks generated will be within a certain range of *S*? A lower bound may not be so important as if the number of tasks generated is not

large enough, dynamic work-stealing would help keep the processors busy. On the other hand, an upper bound may be more important as we want to avoid degrading the performance of the parallel algorithm by generating too many tasks.

7.2.5. Writing POET Patterns

More work needs to be done in developing methods that make it easier for POET users to construct a pattern. Most developers diagnosing faults would find it difficult to translate a fault (such as a performance bottleneck) into a POET pattern. One might think of developing a higher-level language that can be easily understood by POET users and can be translated into the current pattern language. A starting point may be to identify common faults (such as performance bottlenecks or race conditions) and see if suitable techniques can be found to map subsets of these to POET patterns.

Appendix A

In the grouped-approach implementation discussed in Section 4.3.3, we mentioned the importance of adjusting the group size, *g* when generating tasks. We introduced a variable λ such that when *g* is above $\lambda * |D_d|$, it is adjusted to the domain size of the variable at depth level *d*, i.e., $|D_d|$; otherwise, it is adjusted to half of $|D_d|$. Here we show the mathematical relationship between λ, the actual number of tasks generated, *S'*, and the initial desired number of tasks, *S*.

Equation (1) represents the actual number of tasks generated when *g* is adjusted to half of $|D_d|$, where M is the total number of nodes at level *d*.

$$S' = \frac{2M}{|D_d|} \qquad (1)$$

Equation (2) represents the initial desired number of tasks. Note that the exact value of *S* is the ceiling of M/g, but this has been omitted from the equation for simplicity.

$$S = \frac{M}{g} \qquad (2)$$

Let σ be a factor that represents by how much *S'* is greater than *S*, i.e.,

$$S' = \sigma S \qquad (3)$$

Note that the maximum value of *g* for Equation (1) to hold is $\lambda * |D_d|$; when *g* is above this threshold *S'* is $M/|D_d|$. Substituting this maximum value, as well as Equations (1) and (2) in (3), gives Equation (4).

$$\frac{2M}{|D_d|} = \frac{\sigma M}{\lambda |D_d|} \qquad (4)$$

When Equation (4) is resolved we get Equation (5).

$$\sigma = 2\lambda \qquad (5)$$

In the scattered approach, the initial task size, *S* (see Equation 6) is increased when it is greater than $\lambda*|D_d|$. Equation (7) shows the new task size *S'* in this scenario. Substituting both equations in Equation (3) gives Equation (8). The scattered approach is opposite to the grouped one in that when *S* is greater than $\lambda*|D_d|$, the new task size is greater.

$$S > \lambda |D_d| \qquad (6)$$

$$S' = |D_d| \qquad (7)$$

$$\sigma = 1/\lambda \qquad (8)$$

From equations (5) and (8), we see that the optimum value of λ that minimizes σ for both the grouped and scattered approaches is given in equation (9). This value can be used in the task-generation algorithm or λ can be set to some other value taking into consideration the initial desired tasks size, the number of cores available, and the task-generation approach employed.

$$(\sigma = 1/\lambda = 2\lambda) \Rightarrow \lambda = \sqrt{0.5} \approx 0.707 \qquad (9)$$

Appendix B

The following shows the re-written pattern for the “FirstConnectionEstablished” pattern. It consists of 14 disjunctions. The second to eighth disjunctions are the same except for the last happens-before pair. The same holds for the ninth to twelfth disjunctions.

<table>
<tr><td>1</td><td>(*sc_all !→ $sc1 | *sc_all !→ *dc | $dc → *sc_all | $sc2 → *sc_all | $sc1 → *dc | $dc → *dc | $sc2 → *dc | *dc !→ $sc2) &</td></tr>
<tr><td>2</td><td>($sc1 → *dc | *sc_all !→ $sc1 | *sc_all !→ *dc | $dc → *sc_all | $dc → *dc | $sc2 → *sc_all | $sc2 → *dc | *dc !→ $dc) &</td></tr>
<tr><td>3</td><td>($sc1 → *dc | *sc_all !→ $sc1 | *sc_all !→ *dc | $dc → *sc_all | $dc → *dc | $sc2 → *sc_all | $sc2 → *dc | *sc_all !→ $sc2) &</td></tr>
<tr><td>4</td><td>($sc1 → *dc | *sc_all !→ $sc1 | *sc_all !→ *dc | $dc → *sc_all | $dc → *dc | $sc2 → *sc_all | $sc2 → *dc | *sc_all !→ $dc) &</td></tr>
<tr><td>5</td><td>($sc1 → *dc | *sc_all → $sc1 | *sc_all !→ *dc | $dc → *sc_all | $dc → *dc | $sc2 → *sc_all | $sc2 → *dc | *dc !→ $sc2) &</td></tr>
<tr><td>6</td><td>($sc1 → *dc | *sc_all → $sc1 | *sc_all !→ *dc | $dc → *sc_all | $dc → *dc | $sc2 → *sc_all | $sc2 → *dc | *dc !→ $dc) &</td></tr>
<tr><td>7</td><td>($sc1 → *dc | *sc_all → $sc1 | *sc_all !→ *dc | $dc → *sc_all | $dc → *dc | $sc2 → *sc_all | $sc2 → *dc | *sc_all !→ $sc2) &</td></tr>
<tr><td>8</td><td>($sc1 → *dc | *sc_all → $sc1 | *sc_all !→ *dc | $dc → *sc_all | $dc → *dc | $sc2 → *sc_all | $sc2 → *dc | *sc_all !→ $dc) &</td></tr>
<tr><td>9</td><td>(*sc_all → $sc1 | *sc_all !→ *dc | $dc → *sc_all | $dc → *dc | $sc2 → *sc_all | $sc2 → *dc | *dc !→ $sc2) &</td></tr>
<tr><td>10</td><td>(*sc_all → $sc1 | *sc_all !→ *dc | $dc → *sc_all | $dc → *dc | $sc2 → *sc_all | $sc2 → *dc | *dc !→ $dc) &</td></tr>
<tr><td>11</td><td>(*sc_all → $sc1 | *sc_all !→ *dc | $dc --> *sc_all | $dc --> *dc | $sc2 --> *sc_all | $sc2 → *dc | *sc_all !→ $sc2) &</td></tr>
<tr><td>12</td><td>(*sc_all → $sc1 | *sc_all !→ *dc | $dc → *sc_all | $dc → *dc | $sc2 → *sc_all | $sc2 → *dc | *sc_all !→ $dc) &</td></tr>
<tr><td>13</td><td>($sc2 → $dc) &</td></tr>
<tr><td>14</td><td>(*sc !→ $dc | $sc2 !→ *sc)</td></tr>
</table>

References

[1] Eclipse.org home. http://www.eclipse.org.

[2] HSQLDB home. http://www.hsqldb.org.

[3] mySQL home. http://www.mysql.com.

[4] MPI home. http://www.mcs.anl.gov/research/projects/mpi/

[5] PostgreSQL home. http://www.postgresql.org.

[6] Saswat Anand, Wei-Ngan, and Chin Siau-Cheng Khoo. A lazy divide and conquer approach to constraint solving. In *Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence*, pages 91-98, Washington, DC, USA, 2002.

[7] Fahiem Bacchus. Extending forward checking. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, pages 35–51, Singapore, September 2000.

[8] Dwight S. Bedasse. *An Efficient Computation of Convex Closure on Abstract Events.* Master's thesis, University of Waterloo, Department of Electrical and Computer Engineering, Waterloo, Ontario, Canada, 2004.

[9] Gary S. Bloom and Solomon W. Golomb. Applications of numbered unidirected graphs. In *Proceedings of the IEEE,* pages 562–570, April, 1977.

[10] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, New Mexico, USA, November 1994.

[11] Lucas Bordeaux, Youssef Hamadi, and Horst Samulowitz. Experiments with massively parallel constraint solving. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 443-448, California, USA, July 2009.

[12] Andrei A. Bulatov. Tractable conservative constraint satisfaction problems. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, page 321-330, Ottawa, Canada, June 2003.

[13] Wing H. Cheung. *Process and Event Abstraction for Debugging Distributed Programs*. PhD thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1989.

[14] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, pages 226-241, Lisbon, Portugal, September 2009.

[15] Colin J. Fidge. Logical time in distributed computing systems. In *IEEE Computer*, vol. 24(8), pages 28-33, August 1991.

[16] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *USENIX Annual Technical Conference*, pages 27–27, Boston, USA, May 2006.

[17] Matthew L. Ginsberg. Dynamic backtracking. In *Journal of Artificial Intelligence Research*, vol. 1, pages 25-46, August 1993.

[18] Zineb Habbas, Micahel Krajecki, and Daniel Singer. Parallelizing combinatorial search in shared memory. In *Proceedings of the 4th European Workshop on OpenMP*, pages 1-14, Roma, Italy, September 2002.

[19] Zineb Habbas, Michael Krajecki and Daniel Singer. Domain decomposition for parallel resolution of constraint satisfaction problems with OpenMP. In *Proceedings of the 2nd European Workshop on OpenMP*, Edinburgh, Scotland, UK, September 2000.

[20] Zineb Habbas, Michael Krajecki, and Daniel Singer. The Langford's Problem: A challenge for parallel resolution of CSP. In *Proceedings of the 4th International Conference on Parallel Processing and Applied Mathematics*, pages 789-797 Naleczow Poland, September 2001.

[21] Alex Ho and Steven Hand. On the design of a pervasive debugger. In *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging*, pages 117–122, California, USA, September 2005.

[22] Christian E. Jaekl. *Event-Predicate Detection in the Debugging of Distributed Applications.* Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1996.

[23] George Katsirelos and Fahiem Bacchus. Unrestricted nogood recording in CSP search. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, pages 873-877, Kinsale, Ireland, September 2009.

[24] Michael Krajecki, Christophe Jaillet and Alain Bui. Parallel tree search for combinatorial problems: A comparative study between OpenMP and MPI. In *Studia Informatica Universalis*, pages 151-190, December 2005.

[25] Thomas Kunz. *Abstract Behaviour of Distributed Executions with Applications to Visualization.* PhD thesis, Fachbereich Informatik, Technische Hochschule, Darmstadt, 1994.

[26] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, pages 558-565, July 1978.

[27] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek and Zheng Zhang. D3S: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 423–437, San Francisco, USA, April 2008.

[28] Nicolas Loriant and Jean-Marc Mernaud. The case for distributed execution replay using a virtual machine. In *Proceedings of the 15th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 181-186, Manchester, UK, June 2006.

[29] Friedemann Mattern. Virtual time and global states in distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215-226, Chateau de Bonas, France, October 1989.

[30] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Parallelizing constraint programs transparently. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, *Lecture Notes in Computer Science*, pages 514–528, Rhode Island, USA, September 2007.

[31] Ugo Montanari. Network of constraints: Fundamental properties and applications to picture processing. In *Information Sciences*, vol. 7, pages 95–132, 1974.

[32] Matthew Nichols. *Efficient Pattern Search in Large Partial-Order Data Sets*. PhD thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 2008.

[33] Vasco Pedro and Salvador Abreu. Distributed work stealing for constraint solving. In *Online Proceedings of the Joint Workshop on Implementation of Constraint Logic Programming Systems and Logic-based Methods in Programming Environments*, Edinburgh, Scotland, U.K., July, 2010.

[34] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffery C. Mogul, Mehul A. Shah and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *3rd USENIX Symposium on Networked Systems Design and Implementation*, pages 115–128, California, USA, May 2006.

[35] Ilene Seeleman. *Application of Event-Based Debugging Techniques to Object-Oriented Executions.* Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1995.

[36] Michiel F. H. Seuren. *Design and Implementation of an Automatic Event Abstraction Tool*. Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1996.

[37] Marius-calin Silaghi and Boi Faltings. Parallel proposals in asynchronous search. Technical Report (TR-01/371), Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, August 2001.

[38] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, pages 157 -180, 1993.

[39] Ping Xie. *Convex-Event based Offline Event-Predicate Detection.* Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 2003.

[40] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 229-244, Massachusetts, USA, April, 2009.

[41] Yuh M. Yong and David J. Taylor. Performing replay in an OSF DCE environment. In *Proceedings of the 1995 CAS Conference. IBM Canada Ltd. Laboratory, Centre for Advanced Studies*, pages 52-62, Toronto, Canada, November 1995.

[42] Yuh M. Yong. *Replay and Distributed Breakpoints in an OSF DCE Environment*. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada. June 1995.