A Parallel Workflow for Online Correlation and Clique-finding
With Applications to Finance

by
Camilo Rostoker

B.Sc., University of Regina, 2004

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in
The Faculty of Graduate Studies
(Computer Science)

The University Of British Columbia
February, 2007
© Camilo Rostoker 2007
Abstract

This thesis investigates how a state-of-the-art Stochastic Local Search (SLS) algorithm for the maximum clique problem can be adapted for and employed within a fully distributed parallel workflow environment. First we present parallel variants of Dynamic Local Search (DLS-MC) and Phased Local Search (PLS), demonstrating how a simple yet effective multiple independent runs strategy can offer superior speedup performance with minimal communication overhead. We then extend PLS into an online algorithm so that it can operate in a dynamic environment where the input graph is constantly changing, and show that in most cases trajectory continuation is more efficient than restarting the search from scratch. Finally, we embed our new algorithm within a data processing pipeline that performs high-throughput correlation and clique-based clustering of thousands of variables from a high-frequency data stream. For communication within and between system components, we use MPI, the de-facto standard API for message passing in high-performance computing. We present algorithmic and system performance results using synthetically generated data streams, as well as a preliminary investigation into the applicability of our system for processing high-frequency, real-life intra-day stock market data in order to determine clusters of stocks exhibiting highly correlated short-term trading patterns.
# Table of Contents

Abstract .................................................. ii

Table of Contents ........................................ iii

List of Tables ............................................. vi

List of Figures ............................................. viii

1 Introduction ............................................. 1

2 Background .............................................. 6
   2.1 Network Models of Complex Systems .................. 6
   2.2 Graph Theory Basics .................................. 8
   2.3 The (Unweighted) Maximum Clique and Maximum Independent Set Problems ............. 10
   2.4 Stochastic Local Search for Combinatorial Optimization .......................... 11

3 Related Work ............................................ 14
   3.1 Stochastic Local Search for the Maximum Clique Problem ....................... 14
   3.2 Applications Reducing to the Maximum Clique Problem ......................... 19
   3.3 Parallel Metaheuristics ................................ 20
   3.4 Stochastic Local Search in Dynamic Environments ................................. 23
   3.5 Online Graph Algorithms for Dynamic Environments ............................ 26
   3.6 Computational and High Frequency Finance ...................................... 30

4 Parallel Dynamic Local Search and Phased Local Search .................. 32
   4.1 Independent Multiple Runs ................................ 33
      4.1.1 Overview ........................................... 33
      4.1.2 Implementation Considerations .......................... 34
      4.1.3 Cost Model ........................................ 36
   4.2 Cooperative Search Strategy ................................ 39
   4.3 Empirical Analysis ..................................... 42
Table of Contents

4.4 Results .................................................. 47  
  4.4.1 PPLS and PDLS-MC .................................. 47  
  4.4.2 CPPLS .................................................. 50  
4.5 Summary .................................................. 52  

5 Online Phased Local Search .................................. 64  
  5.1 From Offline to Online .................................. 64  
  5.2 Implementation Details .................................. 68  
  5.3 Adding Edges ............................................. 70  
  5.4 Removing Edges .......................................... 71  
  5.5 Empirical Analysis ....................................... 74  
  5.6 Results .................................................. 77  
    5.6.1 Synthetic Dynamic Graph Series ..................... 78  
    5.6.2 Mann-Whitney U Test ................................ 80  
    5.6.3 Stock Market Dynamic Graph Series ................. 81  
    5.6.4 Discussion .......................................... 82  
5.7 Summary .................................................. 83  

6 The Parallel Workflow ........................................ 92  
  6.1 System Design and Architecture ......................... 93  
    6.1.1 Inter-process Communication ......................... 95  
    6.1.2 Pipeline Architecture ............................... 95  
    6.1.3 Processor Farm Architecture ......................... 95  
    6.1.4 Mapping Processes to Processors ................... 96  
  6.2 Time Series Data ......................................... 97  
  6.3 Correlation Calculation .................................. 99  
    6.3.1 Maintaining the Real-time Correlation Matrix ....... 99  
    6.3.2 Communication of Correlation Updates ............... 101  
  6.4 Clique-based Clustering ................................ 101  
    6.4.1 Thresholding ......................................... 101  
    6.4.2 Online Parallel PLS ................................ 101  
    6.4.3 Recording Sets of Maximal Cliques ................. 102  
    6.4.4 Polling for Messages ................................ 103  
  6.5 Evaluation ................................................ 104  
  6.6 Summary .................................................. 106  

7 A Real-time Stock Market Application ....................... 109  
  7.1 Technical Analysis ...................................... 110  
  7.2 Stock Market Time Series Data ......................... 111  
    7.2.1 Historical Data ..................................... 111
# Table of Contents

7.2.2 Intra-day Data ........................................ 112
7.2.3 Working With Multiple Time Series .................. 114
7.3 Technical Indicators as a Similarity Measure ........... 117
7.4 A Parallel Workflow for High-Frequency Stock Market Data 118
  7.4.1 Computing Environment ............................ 120
  7.4.2 Correlation Calculation ............................. 120
  7.4.3 Maintaining the Dynamic Intra-day Market Graph .... 121
  7.4.4 Clique-based Clustering ............................. 121
  7.4.5 A Prototype Visualization Client .................... 122
7.5 Evaluation ........................................... 125
  7.5.1 Homogeneous Time Series ............................ 126
  7.5.2 Inhomogeneous Time Series .......................... 129
7.6 Potential Applications .................................. 130
7.7 Summary ............................................. 133

8 Conclusion ............................................. 135

Bibliography ............................................. 141

Appendices

A Scalability and Speedup Results for PPLS, PDL5-MC and CPPLS .............................................. 153

B Performance Results for OPPLS ........................................ 166
List of Tables

4.1 Values for $T_s$ and $T_p$ determined on our reference machine on each of our four benchmark instances; see text for details. . . 38
4.2 Estimated run-time (Est), actual run-time (Act), and estimation error (Err) for DLS-MC and PLS on our four benchmark instances using the cost model defined in Equation 4.1.3. . . 38
4.3 Properties of our selected DIMACS benchmark instances. . . 43
4.4 Observed speedup when starting with unique seed vertices using 13 search processes. . . . 48
4.5 Speedup and Efficiency for PDLS-MC and PPLS relative to the corresponding sequential versions. . . . 49
5.1 Search step improvement using trajectory continuation on additive series. . . 79
5.2 Search step improvement using trajectory continuation on subtractive series. . . 79
5.3 Search step improvement using trajectory continuation on mixed series. . . 79
5.4 Number of stage transitions in which the maximum clique size changes. . . 79
5.5 Percentage of stages for which the null hypothesis was rejected with $\alpha = 0.5$. . . 81
5.6 Performance differences between PLS with and without TC on the dynamic market graph series. . . 81
6.1 Speedup for workflow environment with 2000 stocks and batch size 2000. . . 106
6.2 Response time statistics for varying processor configurations. 107
7.1 One second of sample quote data from the TSX. . . . 113
7.2 Quote frequency distributions from our sample TSX quote data. $Q(0.9)$ and $Q(0.1)$ are the 0.9 and 0.1 quantiles, respectively . . . 115
List of Tables

7.3 Hypothetical correlation matrix for five stocks. . . . . . . . 121
List of Figures

2.1 A graph with a maximum clique and maximum independent set of size 4. 9

3.1 A sample graph showing $K$, $C_0(K)$ and $C_1(K)$. 19

3.2 A star cover before (A) and after (B) a new vertex is added. 29

4.1 Flowchart of multiple independent runs parallelization strategy. 35

4.2 Two sample scatter plots for PPLS, showing the correlation between run-length (x-axis) and run-time (y-axis). 40

4.3 RLD and RTDs for the C1000.9 instance using 16 search processes. 45

4.4 Sample Scalability and Speedup results for PPLS. Plot (a) shows for varying numbers of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length. 54

4.5 Sample Scalability and Speedup for PDL-MC. Plot (a) shows for varying number of processors the RLDs while (b) shows the corresponding speedup based on median run-length. 55

4.6 Empirical RLDs and corresponding approximated RLDs with exponential distribution for PDL-MC on the keller6 instance. 56

4.7 Sample Scalability and Speedup results for CPPLS. Plot (a) shows speedup based on the median run-length, while plot (b) shows speedup based on the median run-time. 57

4.8 A surprising Scalability and speedup result for CPPLS. Plot (a) shows speedup based on the median run-length, while plot (b) shows speedup based on the median run-time. 58

4.9 Comparing RLDs for PPLS and CPPLS with 1 search process on two of our four benchmark instances. 59

4.10 Comparing Scalability (left) and Speedup (right) of CPPLS to PPLS. Plot (a) shows speedup with respect to the median run-length, while plot (b) shows speedup with respect to the median run-time. 60
List of Figures

4.11 Comparing Scalability (left) and Speedup (right) of CPPLS to PPLS. Plot (a) shows speedup with respect to the median run-length, while plot (b) shows speedup with respect to the median run-time. ............................................ 61

4.12 Impact of the value of MAX_ELITE parameter on CPPLS, applied to the keller6 problem instance. ............................................ 62

4.13 Impact of the value of \( \beta \) parameter on the RTD of CPPLS, applied to the keller6 problem instance over two search process configurations. The three \( \beta \) parameter values are 50, 100 and 200; the CPPLS RTDs are compared against the PPLS RTDs with 2 and 8 search processes respectively. ............................................ 63

5.1 RLDs (top) and RTDs (bottom) for PLS with and without the sparse graph enhancement. ............................................ 71

5.2 Maximum clique size in the brock200.1 mixed dynamic graph series with \( n = 40 \). ............................................ 76

5.3 Speedup observed when using Trajectory Continuation with various dynamic graph series. ............................................ 78

5.4 Online performance results for the p_hat500-1 mixed series with \( n = 10 \); the top figure is selections per graph, the bottom figure is cumulative selections. ............................................ 84

5.5 Online performance for the p_hat500-1 mixed series with \( n = 20 \); the top figure is selections per graph, the bottom figure is cumulative selections. ............................................ 85

5.6 Online performance for the p_hat500-1 mixed series with \( n = 40 \); the top figure is selections per graph, the bottom figure is cumulative selections. ............................................ 86

5.7 Two different but typical RLDs for the additive series with \( n = 20 \). ............................................ 87

5.8 Typical RLDs for the subtractive series with \( n = 20 \). ............................................ 88

5.9 Maximum clique size trace through the dynamic market graph series. ............................................ 89

5.10 Edge count and edge density as a function of correlation threshold. ............................................ 90

5.11 Vertex degree distributions on the dynamic market graph series for correlation thresholds \( t = \{0.3, 0.4, 0.5, 0.6\} \). ............................................ 91

6.1 Topology of process structure and communication model ............................................ 94

6.2 Processor farm architecture and communication timeline. ............................................ 96

6.3 Data sample queues for all \( N \) variables at time \( t \) ............................................ 98
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4</td>
<td>Communication timeline between Maronna and OPPLS.</td>
</tr>
<tr>
<td>6.5</td>
<td>poll_interval trace with using a 5 second sampling interval.</td>
</tr>
<tr>
<td>6.6</td>
<td>Response time for a dataset with 2000 variables, batch size 2000, and three processor configurations.</td>
</tr>
<tr>
<td>7.1</td>
<td>Two inhomogeneous time series with OHLC format. Each 'Q' represents a single quote.</td>
</tr>
<tr>
<td>7.2</td>
<td>An instance of the parallel workflow for real-time stock market analysis.</td>
</tr>
<tr>
<td>7.3</td>
<td>Constructing thresholded market graphs from a complete market graph.</td>
</tr>
<tr>
<td>7.4</td>
<td>Snapshot of a market graph with positive correlations.</td>
</tr>
<tr>
<td>7.5</td>
<td>Energy stocks showing correlated QFSMA indicators with 3 minute sampling interval.</td>
</tr>
<tr>
<td>7.6</td>
<td>Energy stocks showing correlated QFSMA indicators with 1 minute sampling interval.</td>
</tr>
<tr>
<td>7.7</td>
<td>Resource and other stocks showing correlated QFSMA indicators with 1 minute sampling interval.</td>
</tr>
<tr>
<td>7.8</td>
<td>Gold stocks showing correlated QVSMA indicators with a 30 second sampling interval.</td>
</tr>
<tr>
<td>7.9</td>
<td>Correlated QFSMA indicators over a 2.5 minute time window with a 3 second sampling interval.</td>
</tr>
<tr>
<td>7.10</td>
<td>Three pair-wise correlation scatter plots for the stocks in Figure 7.9. The first company name corresponds to the values on the X-axis, the second name to the Y-axis.</td>
</tr>
<tr>
<td>7.11</td>
<td>Correlated event-based QMROC patterns.</td>
</tr>
<tr>
<td>7.12</td>
<td>The underlying time series for the QMROC events in Figure 7.11.</td>
</tr>
<tr>
<td>A.1</td>
<td>Scalability and Speedup results for PPLS on the brock800.1 instance. Plot (a) shows for varying numbers of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length.</td>
</tr>
<tr>
<td>A.2</td>
<td>Scalability and Speedup results for PPLS on the p_hat1500-1 instance. Plot (a) shows for varying numbers of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length.</td>
</tr>
</tbody>
</table>
A.3 Scalability and Speedup results for PPLS on the C1000.9 instance. Plot (a) shows for varying numbers of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length .................................................. 156

A.4 Scalability and Speedup results for PPLS on the keller6 instance. Plot (a) shows for varying numbers of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length .................................................. 157

A.5 Scalability and Speedup results for PDLS-MC on the brock800.1 instance. Plot (a) shows for varying number of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length .................................................. 158

A.6 Scalability and Speedup results for PDLS-MC on the p.hat1500-1 instance. Plot (a) shows for varying number of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length .................................................. 159

A.7 Scalability and Speedup results for PDLS-MC on the C1000.9 instance. Plot (a) shows for varying number of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length .................................................. 160

A.8 Scalability and Speedup results for PDLS-MC on the keller6 instance. Plot (a) shows for varying number of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length .................................................. 161

A.9 Scalability and Speedup results for CPPLS on the brock800.1 instance. Plot (a) shows speedup based on the median run-length, while plot (b) shows Speedup based on the median run-time ........................................................................ 162

A.10 Scalability and Speedup results for CPPLS on the p.hat1500-1 instance. Plot (a) shows speedup based on the median run-length, while plot (b) shows Speedup based on the median run-time ........................................................................ 163

A.11 Scalability and Speedup results for CPPLS on the C1000.9 instance. Plot (a) shows speedup based on the median run-length, while plot (b) shows Speedup based on the median run-time ........................................................................ 164

A.12 Scalability and Speedup results for CPPLS on the keller6 instance. Plot (a) shows speedup based on the median run-length, while plot (b) shows Speedup based on the median run-time ........................................................................ 165

xi
List of Figures

B.1 Online Performance for phat500-1 20-stage Additive Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. 167

B.2 Online Performance for brock200.1 20-stage Additive Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections 168

B.3 Online Performance for keller4 20-stage Additive Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. 169

B.4 Online Performance for p_hat500-1 20-stage Subtractive Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. 170

B.5 Online Performance for brock200.1 20-stage Subtractive Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections 171

B.6 Online Performance for keller4 20-stage Subtractive Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections 172

B.7 Online Performance for p_hat500-1 10-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. 173

B.8 Online Performance for brock200.1 10-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. 174

B.9 Online Performance for keller4 10-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. 175
List of Figures

B.10 Online Performance for p_hat500-1 20-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. . 176

B.11 Online Performance for brock200_1 20-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. . 177

B.12 Online Performance for keller4 20-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections 178

B.13 Online Performance for p_hat500-1 40-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. . 179

B.14 Online Performance for brock200_1 40-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. . 180

B.15 Online Performance for keller4 40-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections 181

B.16 Online Performance for the Additive Dynamic Market Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. . . 182

B.17 Online Performance for the Subtractive Dynamic Market Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. . . . . 183

B.18 Online Performance for the Mixed Dynamic Market Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections. . . . . 184

xiii
Chapter 1

Introduction

Each day around the world, dozens of stock exchanges combined are hosting tens of thousands of stocks, executing millions of trades, processing and matching billions of quotes, with total dollar values of more than a hundred billion dollars. The NASDAQ alone, one of the biggest exchanges in the world, averaged 66 million quotes, 6.5 million trades, 1.6 billion shares executed, and $51 billion in dollar volume over first three quarters in 2006 [1]. At the heart of these exchanges are powerful electronic transaction systems; the sheer frequency of the input data is so enormous that they require specialized algorithms and data-processing techniques. Moreover, within this high-frequency stream of data lies hidden information, from which meaningful economical knowledge can be extracted. For this reason, brokerage firms are racing to continuously improve their complex financial models, gearing them towards analyzing market data in real-time, to find patterns and trends in order to predict price movements, with the bottom-line objective of maximizing their clients return on investment (ROI). Data-mining and computational intelligence practitioners have, for some time, been using statistical analysis and advanced algorithmic techniques to analyze the massive amount of financial data. Such methods are useful for finding valuable information in large static datasets, with applications in portfolio and risk management, money-laundering detection, credit rating analysis and long-term trading rule selection [2].

Due to the recent shift away from traditional (physical) stock markets to electronic markets (partially or completely), the ability to perform real-time computational analysis has become a major focus for many financial institutions. The evidence is a growing number of high-throughput electronic exchanges offering real-time, low-latency data feeds. The result is an electronic trading environment where in order to out-compete, you not only need to out-smart, but you also need to out-compute. It is already accepted in the industry that both computational speed and computational intelligence will be critical components for any successful next-generation financial trading platform [3, 4]. At the heart of this transition is the new field of High Frequency Finance, a systematic and algorithmic approach to analyzing,
Chapter 1. Introduction

modeling and inference of high-frequency financial time series — the source of the price formation process [3]. Results from the past decade’s worth of research on high-frequency finance has formed the foundation for Automated and Algorithmic Trading. While often used interchangeably, “Automated Trading” refers to the action of automatically deciding what to buy/sell, whereas Algorithmic Trading usually specifies how to buy/sell a chosen stock most efficiently; e.g., breaking a large order into multiple smaller orders to minimize the market impact. These new trading systems depend on the ability to process and analyze high-frequency, multi-dimensional and inhomogeneous time-series data streams as quickly and efficiently as possible, in an on-line fashion. Unfortunately, like in other real-world problems, the issue of data processing and analysis becomes increasingly complex as the size and frequency of data increases. As a result, many new challenges and opportunities exist in the area of high-performance computing for real-time data acquisition and processing.

With that in mind, the goal of this thesis was to develop a system that is capable of processing such high-frequency financial data streams in real-time. The result is a data processing pipeline composed of multiple computing components working in parallel. To explore the feasibility of predicting short-term movements, we have proposed a system that performs correlation and clustering of the input data (e.g., real-time stock market trade and quote orders). The core components in our system perform correlation and clustering, both of which are computationally intensive tasks. In order to perform these tasks in real-time, we use online and parallel versions of Maronna, a robust correlation method, and Phased Local Search (PLS), a graph-theoretic clustering method. The output from the system are subsets of stocks showing highly correlated (or anti-correlated) short-term activity.

Since processor speeds have reached their limits, two trends are emerging as the main approaches for increasing overall computing speeds. Compute clusters — often referred to as Beowulf clusters when assembled from off-the-shelf hardware and software — are attractive for small-to-medium sized companies who need access to such high-performance computing (HPC) infrastructures, but cannot afford the large-scale supercomputers offered by the major supercomputing vendors such as IBM, SGI and Cray. More recently, multi-core processors for the consumer PC have emerged, indicating yet another opportunity for commodity hardware to be used within an HPC context. On the software side, many open-source projects for parallel computing and HPC are gaining market share in the wake of this hardware paradigm shift. For example, consider MPI (Message Passing Interface), the de-facto API for message passing applications in high-performance com-
puting [5]. Due to the openness and wide-spread usage of these projects, MPI runs over almost all kinds of network infrastructures including TCP/IP, allowing the system components to be effortless distributed over a wide-area network (WANs). While MPI has seen widespread usage for over 10 years in academia and specialized research centers, various open-source implementations of MPI such as LAM [6], OpenMPI [7] and MPICH [8] are continuously improving, offering a cost-effective alternative to parallel computing.

It is believed that the stock market is a complex system, continuously evolving with every new piece of information [9]. By collecting massive amounts of data from the stock market, researchers are able to empirically measure the intrinsic properties and behavioral characteristics of the system that may otherwise go unnoticed by the passive observer. A general method of modeling complex systems is by means of a network: a set of vertices (or nodes) connected to each other by edges (or links). The exact semantics of the vertices and edges, and the process by which the network is constructed, depends on the system under consideration and which aspects of it are to be modeled. This thesis is motivated by a recent line of research that examines the stock market as a network of interacting components exhibiting complex correlation patterns. A stock market network, often referred to as a market graph, is a model in which stocks are nodes, and links between nodes represent a relationship between the stocks [10]. In most cases, the relationship — herewithin referred to as a similarity measure — is the cross-correlation of the two stocks' log daily price return over a given time period. In this market graph model, the graph is assumed to be static (i.e., edges are not added or removed), and thus the exact configuration and topology of the graph reflects the stock market network over the time period in which the data was collected. Previous research on stock market networks have focused on historical data [10, 11, 12, 13, 14], leading to detailed analyses of the evolution and characteristics of the historical (long-term) market graph.

Different similarity functions will produce vastly different network models. A limitation of the historical market graph is that while it can be used to detect long-term trends, it does not provide much information on the short-term (intra-day) dynamics. An alternative approach is to consider the stock market as a system that continuously evolves over time, from which valuable information can be extracted at any given point. For example, both human day traders and automated trading systems would benefit from knowing which clusters of stocks exhibit highly correlated short-term trading patterns. In this thesis we present an approach which utilizes newly available high-frequency intra-day data to compute and construct the dynamic intra-day market graph, which is essentially a series of consecutive (and re-
lated) graphs representing the evolution of the intra-day market graph. In this model, the high-frequency, dynamic and unpredictable nature of the underlying data demands massive computing power and scalable, online algorithms to ensure real-time responses, since even the smallest unit of information (e.g., a single stock quote) has the potential to change the underlying configuration of the market graph. Due to the unpredictable frequency and distribution of incoming data, re-computing solutions from scratch would not only be a waste of computing resources, but also impose even tighter time constraints in a system where time is already the limiting factor. Our system attempts to address this issue by using online algorithms that are designed to operate efficiently on real-time dynamic data flows, resulting in more accurate and timely information dissemination.

Some of the key questions we wish to explore in this thesis are:

- Which parallel implementations of PLS is most appropriate for use within the system?
- How well do the parallel implementations scale as additional processors are used?
- How can PLS exploit the fact that clique solutions between consecutive graphs are often very similar?
- How fast can we update the dynamic graph and subsequently find the new set of clique solutions?

Similarly, the application of our system to high-frequency financial time series allows us to explore answers to questions such as:

- What are the key issues in maintaining (in real-time) the dynamic intra-day market graph?
- What kind of trends/patterns, if any, exist in the dynamic intra-day market graph?
- If any such patterns exist, how can they be interpreted in a meaningful way?

This thesis addresses these questions by introducing novel parallel and online algorithms that we believe are well-suited for this problem. First, we designed and implemented a parallel version of Phased Local Search (PLS) [15], a Stochastic Local Search (SLS) algorithm for the maximum clique problem, and find it achieves near-optimal speedup using independent
multiple search, a very basic yet highly efficient parallelization strategy. Besides the source code being easily accessible, PLS was chosen because it is state-of-the-art for a wide range of benchmark problem instances. Motivated by our target application, we also extend PLS to find not just the maximum clique, but a large set of maximal cliques within the evolving intra-day market graph. We then designed and implemented an online version of PLS that performs trajectory continuation, a technique that allows PLS to operate in dynamic environments by eliminating the need to restart the search from scratch each time the underlying graph changes. We then take an existing parallel implementation of a robust correlation method called Maronna [16], and introduce minor modifications that allows it to efficiently compute an arbitrary subset of pair-wise correlation coefficients from thousands of high-frequency time-series. Finally, we tie these pieces together within a fully distributed, highly-parallel workflow environment designed to meet the demanding computational requirements outlined above.

Parts of this thesis will appear as a technical paper in the proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium [17]. The remainder of this thesis is organized as follows. Chapter 2 provides basic background material on several Computer Science topics used throughout the thesis, while Chapter 3 discusses related work which motivates much of the research presented in the following chapters. Chapter 4 presents parallel variants of Phased Local Search (PLS) and Dynamic Local Search (DLS-MC), two state-of-the-art SLS algorithms for the maximum clique problem, and in Chapter 5 we introduce our online variant of PLS. Chapter 6 introduces the parallel workflow environment, describes how the data is transformed as it flows through the pipeline, and outlines the communication protocol used by the pipeline components. Chapter 7 describes an application of our system within the context of real-time stock market analysis. Finally, Chapter 8 provides concluding remarks and discusses several exciting avenues for future research.
Chapter 2

Background

This thesis spans several areas in Computer Science including Combinatorial Optimization, Graph Theory, Artificial Intelligence, Parallel Algorithms and High-Performance Computing, and so we devote this chapter to introducing relevant background information that will prepare the reader for the content to follow.

2.1 Network Models of Complex Systems

In a world where data is being generated faster than ever before, there is a need for new and improved data analysis techniques. In many contexts, the data under consideration represents networks of interacting agents or components (also known as complex systems). The general representation of such networks is by means of a graph, which consists of a set of vertices (or nodes) connected to each other by edges (or links). The specific meaning of the nodes and edges, and the process by which the network is constructed, depends on the exact context under consideration. In this section we review several examples of real world datasets that can be accurately represented by a network model.

Perhaps the most famous and intensely studied class of network models are the broad class of social networks, which have been the object of fruitful research in a field referred to as Social Network Analysis (SNA). SNA, in general, aims to rigorously define, analyze and characterize the complex underlying patterns and regularities among interacting units [18]. As most of us have witnessed, humans tend to naturally form friendship networks by organizing into smaller and more manageable groups; when this occurs, everyone in the group knows each other, and a “clique” is formed. This type of social “clustering” is the emergent phenomenon of self-organization, and happens throughout cultures across the world [19]. Other social network representations include co-authorship networks, which represent groups of authors who have written papers or books together [20]. Various corporations, governments and institutions use SNA techniques to analyze their management structure and communication patterns within their organiza-
Military security projects construct and analyze social networks to monitor and identify possible terrorist groups. Online sites such as Friendster (social networking), CiteSeer (scientific digital library) and LinkedIn (personal and professional networking) utilize these social network models to deliver sophisticated and personalized services. Common to all these social networks is that the underlying data is dynamic and complex, requiring state-of-the-art computational methods to efficiently extract interesting and useful information.

In fields such as biology and biochemistry, gene co-regulation and protein-protein interaction networks provide valuable insights into complex biological systems. In the past, such research was mostly based on experiments performed in wet labs, using only a small set of genes (proteins), and would take months and years to complete. Now, in the emerging field of bioinformatics, new computational methods for analyzing network models of biological systems are helping researchers to construct large-scale experiments to determine genome-wide groups of highly correlated genes (or proteins). The main advantage of this new approach is that it provides a global view of the networks under consideration, rather than a small subset of it in isolation, which has not only vastly helped to improve our understanding of known genes and proteins, but also helped to characterize and classify functional activity of previously unknown genes and proteins. We point out that small subsets of genes and proteins are often still analyzed by means of specialized experiments for purposes of validation or elaboration of scientific hypotheses related to the biological processes.

Another network is the World Wide Web (WWW), or Internet. In this network, nodes are web pages while edges between nodes represent hyperlinks between the pages. From the network representation of the WWW a variety of useful information can be extracted. For example, it has been shown that highly connected subsets of nodes in the WWW graph may represent link spam, i.e., websites that attempt to increase their search engine rankings by constructing highly inter-linked groups of similar websites. Another common use is to cluster the network into groups of similar web pages for purposes such as classification, hierarchical organization and general group-based data analysis. Like other network models, the data in this network changes on a daily basis as pages and hyperlinks are updated, added or become obsolete.

Another massive network which continuously produces a large amount of daily data is the so-called call graph, a network whose nodes are phone numbers and edges represent phone calls between the two numbers. Large telecom companies rely heavily on efficient communication processes within
Chapter 2. Background

their physical network in order to optimize performance and minimize overhead costs. Several recent papers have analyzed call graphs from AT&T networks containing hundreds of millions of nodes and billions of edges [32, 33]. An important aspect of this model is that it can be constructed to model the short-term structure of the call network; i.e., edges are added to the graph as calls are made while existing edges representing calls older than a specified time period are removed.

Finally, the finance industry has taken a keen interest in modeling the stock market as a complex network [10, 11, 12, 13], with recent findings suggesting that it exhibits non-random properties [14] and is composed of many interacting subsystems [34]. Due to the recent shift away from traditional (physical) stock markets to (partially or completely) electronic markets, massive amounts of detailed intra-day trading data is now easily accessible. A stock market network, often referred to as a market graph [10], is a network model in which nodes are stocks and links between nodes represent some relationship between the stocks. In most studies, the relationship is the cross-correlation of a pair of stocks log daily price returns over a specified time period. Valuable information can be extracted from the market graph using standard graph theory analysis (see Section 2.2). For example, a fully connected subset of the market graph represents stocks which tend to have correlated (or anti-correlated) price trajectories, suggesting they have similar risk profiles. Similarly, a subset of stocks with few or no edges between them would suggest they are highly non-correlated, thus representing a well-diversified portfolio. Such information is valuable for constructing portfolios with optimal sector allocations and hedging strategies, but also to maintain diversification in order to avoid exposure to excessive risk.

2.2 Graph Theory Basics

In this section we briefly review some graph theory concepts used throughout the remainder of this thesis.

Given an undirected, unweighted graph \( G = (V, E) \), \( V \) is the set of vertices and \( E \) is the set of edges. The common convention is to label the elements of \( V \) as integers 1 through \( n \), while elements in \( E \) can be specified as a set of pairs of vertices, i.e. \( \{u, v\} \in V \). Furthermore, since \( G \) is undirected, \( \forall u, v, (u, v) \in E \iff (v, u) \in E \). The size of the vertex and edge sets are \( |V| \) and \( |E| \), respectively. We say the degree of a vertex \( v \) is the number of edges incident to it; i.e. \( \text{degree}(v) = d \) implies vertex \( v \) is incident to exactly \( d \) edges. Two nodes are adjacent to each other if they
are incident to the same edge.

The neighbourhood set of a given vertex \( v \) is denoted as \( N(v) \), and represents the set of all vertices which are adjacent to \( v \), i.e., \( N(v) = \{ u \in V \mid (u, v) \in E \} \). Clearly, \( |N(v)| = \text{degree}(v) \). For a subset of vertices \( S \subseteq V \), we define the subgraph induced by \( S \) as \( G(S) = (S, E \cap S \times S) \).

A graph \( G = (V, E) \) is complete if \( \forall i, j \in V, (i,j) \in E, i \neq j \). A clique \( C \) is a subset of \( V \) such that \( G(C) \) is complete. A maximal clique is a clique that is not a subset of any other cliques. A maximum clique of a given graph \( G \) is the clique(s) with greatest cardinality\(^1\). In Figure 2.1 there are many maximal cliques (vertex sets \{3,4,5,6\}, \{1,3,5\} and \{7,8,9\} are a few examples), but there is only one maximum clique, formed by the vertex set \{3,4,5,6\}.

![Figure 2.1: A graph with a maximum clique and maximum independent set of size 4.](image)

An independent set \( I \subset V \) is a set such that \( \forall i, j \in V, (i,j) \notin E \), i.e., there are no edges between any of the vertices in \( I \). In Figure 2.1, vertices \{1,2,7,10\} form an independent set, because there are no edges between them.

Throughout this thesis we work only with unweighted graphs. For context, a weighted graph \( G = (V, E, \omega) \) is one where each edge \((i,j) \in E\) is associated with a weight \( \omega_{ij} \). A weighted graph can be converted to an unweighted one simply by specifying some edge threshold and deleting edges whose weights fall below the threshold. Formally, we can define the thresholding function as \( \text{Thresh}(G, \theta) = (V, E_\theta) \) where \( G = (V, E, \omega) \) is a weighted graph and \( E_\theta = \{ (i,j) \in E \mid \omega_{ij} \geq \theta, i \neq j \} \) is the set of edges whose corresponding weights are greater than or equal to \( \theta \).

\(^1\)There can be multiple maximum cliques with the same cardinality.
Chapter 2. Background

2.3 The (Unweighted) Maximum Clique and Maximum Independent Set Problems

Given an undirected, unweighted graph \( G = (V, E) \), the maximum clique problem is to find in \( G \) a clique with maximum cardinality. The decision variant is to determine if a maximal clique of size \( k \) exists in \( G \). Similarly, the maximum independent set problem is to find in \( G \) an independent set of maximum cardinality, while the decision variant is to determine if a maximum independent set of size \( k \) exists in \( G \). Furthermore, the complement graph of \( G = (V, E) \) is defined as \( \overline{G} = (V, \overline{E}) \), where \( \overline{E} = \{(i,j) \mid i,j \in V, i \neq j \text{ and } (i,j) \notin E\} \). Notice that in \( \overline{G} \), all vertices that were part of an independent set in \( G \) are now fully connected, thus forming a clique. From this observation we can see that the problem of finding a maximum independent set in \( G \) is equivalent to finding a maximum clique in \( \overline{G} \).

The maximum clique and maximum independent set problems have been studied intensely for decades by computer scientists and mathematicians because of its prominent role in graph theory and complexity theory, as well as having many practical applications such as navigation for robotics [35, 36], image recognition [37], air traffic control [38], financial portfolio optimization [10], bioinformatics [26, 27, 28, 29] and sensor network routing [39].

The search variant of the maximum clique and maximum independent set problems are NP-hard while their associated decision problems are NP-complete [40]. Furthermore, it has recently been shown that no deterministic polynomial-time algorithm can find cliques of size \( |V|^{1-\epsilon} \) for any \( \epsilon > 0 \), unless of course \( \mathsf{NP} = \mathsf{ZPP} \) [41].\(^2\) The number of maximal cliques within a graph can grow exponentially in the size of the graph. If \( N = |V| \) is the number of vertices in \( G \), then the upper limit on the number of maximal cliques is \( 3^{N/3} \); e.g., consider a graph with \( N=2000 \) vertices — then there can be up to \( 5.8 \times 10^{317} \) maximal cliques in the graph!

Even though the problem is intractable, years of research has been devoted to complete enumeration algorithms. One of the main enumeration approaches was developed in 1973 by Bron and Kerbosch [42]; this procedure is based on a backtracking method incorporating heuristics to prune the search tree early in the search process. Since then, numerous variations of the algorithm have been proposed. Bomze et al. [43] provide a survey of

\(^2\)This is because the ZPP class of problems can be solved by probabilistic algorithms with zero error probability in expected run-time that is polynomial in the size of the given input.
Chapter 2. Background

the last decade's developments in maximal clique enumeration methods.

In order to perform many of the experiments throughout this thesis, clique enumeration was performed on small graphs (up to 500 vertices) to find the maximum clique size. A version of the Bron and Kerbosch algorithm called MBK [44] was used for clique enumeration since it provided good runtime performance, but also because the author was very helpful by making available the source code and offering useful feedback. The modification introduced in MBK removes the need to choose the next vertex to add to the currently expanding clique (the next branch in the search tree), which runs in quadratic time. Instead, the BK algorithm is used to search each subgraph \( S_i \) (where \( 1 < i < n \)), induced by vertex \( i \) and its neighbors \( N(i) \). This results in a recursive procedure that finds maximal cliques constrained to involve only vertices \( j \geq i \), if the subgraphs \( S_i \) are explored in order of increasing \( i \). For larger graphs, however, we had to resort to using the empirical maximum, which is the best observed solution over many test runs using heuristic search methods. As we can see, clique enumeration methods are severely limited by the size of the graph. For this reason, numerous heuristic methods have been developed to efficiently solve the maximum clique problem. They are discussed in Section 3.1.

2.4 Stochastic Local Search for Combinatorial Optimization

Informally, combinatorial optimization is defined as "the mathematical study of finding an optimal arrangement, grouping, ordering or selection of discrete objects usually finite in number" [45]. Stochastic Local Search (SLS) is a class of randomized search algorithms for solving complex combinatorial optimization problems (COPs), and are particularly useful in real-time applications, or when knowledge about the problem domain is rather limited [46]. Some well known SLS methods, often (although somewhat imprecisely) referred to as metaheuristics, include Genetic Algorithms, Ant Colony Optimization, Neural Networks, Simulated Annealing and Tabu Search. SLS algorithms, and metaheuristics in general, are extremely well-suited to solve hard COPs from a vast array of application domains such as economics [47], finance [48, 49], vehicle routing [50, 51, 52], network design [53], scheduling [54] and bioinformatics [46, 55].

SLS methods include a wide variety of techniques that exploit certain properties of the underlying search space topology to find the optimal (or close-to-optimal) solutions to a given instance of a combinatorial optimiza-
Chapter 2. Background

In general, an SLS algorithm searches for a solution to a given problem instance in the space of (complete or partial) candidate solutions. The algorithm starts by selecting an initial candidate solution (chosen randomly or by a construction heuristic), and proceeds by moving from one candidate solution to a neighbouring one. At each search step, only a limited amount of local information is used to choose from the set of neighbouring candidate solutions. An evaluation function is often used to determine which of the candidate solutions should be accepted. In the case of the maximum clique problem, an evaluation function can be as simple as comparing the maximum clique size (since the goal is to find the maximal clique of greatest cardinality). In many cases, however, randomized decisions are used to choose from the set of candidate solutions.

All state-of-the-art SLS algorithms implement two important mechanisms within the search strategy, namely intensification and diversification [46]. Intensification is the process of greedily improving the current solution quality within a small area of the search space (neighbourhood) for a local optimum, while diversification is used to prevent stagnation by ensuring that the search process explores a broad region of the search space, rather than being confined to a small area that may contain only suboptimal solutions. Random decisions has proven to be an efficient diversification mechanism, while intensification can be achieved through a variety of techniques such as iterative improvement or the selection step in a genetic algorithm. Furthermore, it has recently been shown that the performance of SLS algorithms are robust to the quality of the underlying random number generator [56].

An important feature of SLS methods (and local search in general) is that they are ideal algorithms for operating in dynamic environments where the problem is continuously changing. The reasons are twofold: first, as already mentioned, SLS algorithms are able to quickly locate promising regions of the solution space, meaning that if the problem changes, a simple restart of the algorithm can quickly rebuild a high quality solution. The second reason is that an SLS algorithm can be altered so that it does not need to restart after the problem changes. The intuition here is that in many dynamic environments, the change is relatively minor and a restart would ignore this fact. Instead, the algorithm makes the necessary adjustments to its internal state and continues the search from the same (or nearby) position in the search space.

While most SLS algorithms are not guaranteed to to find the optimal
solution\textsuperscript{3}, they are able to quickly locate promising regions of the search space containing high-quality solutions — this is a very desirable characteristic for real-time systems with tight time constraints. To emphasize this point, let us contrast SLS methods with complete (exhaustive) methods which systematically search the entire space of possible solutions. While these approaches are usually guaranteed to find the optimal solution (if one exists), early decisions often lead the search into sub-optimal regions of the solution space, thus making them more susceptible to finding poor solutions in time constrained contexts.

\textsuperscript{3}Many SLS algorithms are Probabilistically Approximately Complete (PAC), meaning that if allowed to run long enough, the probability of finding the optimal/target solution converges to 1.0 [46].
Chapter 3

Related Work

This chapter reviews related work from the various areas of Computer Science touched upon in this thesis.

3.1 Stochastic Local Search for the Maximum Clique Problem

A problem posing many challenges for graph-based algorithms is the sheer size and complexity of the networks under consideration. As outlined in Section 2.1, a growing number of datasets are being modeled as networks, with many such networks containing millions of vertices and edges. Complete enumeration of maximal cliques for the purpose of finding the maximum clique is not a feasible approach for these large problems, and even for many smaller problems if there are stringent time constraints. As a result, many heuristic algorithms have been developed to address this issue. We now review several of the more recent state-of-the-art approaches.

Following the success of the Reactive Tabu Search framework [57], Battiti developed Reactive Local Search (RLS) [58], a reactive SLS algorithm for the maximum clique problem which complements local search with a prohibition-based diversification mechanism. The amount of diversification is dynamically controlled via a history-sensitive feedback scheme which is able to detect and deter cycles in the search trajectory. Since its inception, RLS has enjoyed much attention and for many years served as one the best performing heuristic for the maximum clique problem. Although more recent algorithms have been developed that outperform RLS on many benchmark instances, especially for large and harder instances such as the Brockington-Culberson [59] and Keller [60] graphs, it remains an important high-performance algorithm that is often used in competitive studies.

Grosso and Locatelli proposed Deep Adaptive Greedy Search (DAGS) [61], which aims to combine the simple yet efficient GRASP (Greedy Randomized Adaptive Search Procedure) approach with concepts from the “restart-adaptive” methods proposed by Jagota and Sanchis [62]. In short, DAGS
is a two-phase heuristic: the first phase uses a multistart improving greedy procedure which builds a number of good cliques and then scores nodes by the number of cliques they belong; the second phase is based on a pure greedy approach which uses the weights obtained in the first phase to search “deeper” into the surrounding neighbourhood. DAGS has been shown to be superior to many earlier algorithms, although for some harder instances it fails to compete with RLS [61].

Katayana et al. showed how to efficiently solve the maximum clique problem using k-opt Local Search [63]. Their algorithm is based on Variable Depth Search (VDS) [46], in which each iteration a series of successive add or drop moves are made within a given neighbourhood until no improving solutions can be found. Although k-opt Local Search was shown to perform better than RLS on many DIMACS instances, it is outperformed by Dynamic Local Search and Phased Local Search on nearly all DIMACS instances [15, 64]. The authors, however, are quick to point out that its true potential lies in the ability to integrate it within a more powerful meta-heuristic framework.

Reactive Prohibition-based Ant Colony Optimization (RPACO) [65] combines the intelligent search strategies from Ant Colony Optimization (ACO) with a prohibition-based local search heuristic to solve the maximum clique problem. The main advantage of this approach is that it provides a natural and intuitive parallel formulation, for both deploying a number of ant colonies that search the graph in tandem as well as functionally decomposing the local search procedure. RPACO was shown to be competitive with RLS in terms of the average solution results for a range of DIMACS benchmark instances. However, its run-time performance relative to RLS and other algorithms is unknown.

More recently, Pullan and Hoos proposed Dynamic Local Search (DLS-MC) [64], a stochastic local search algorithm which alternates between phases of iterative improvement, where suitable vertices are added to the current clique, and plateau search, where vertices in the current clique are swapped out and replaced by other vertices. Integer penalties are assigned to vertices as they are considered in the current clique, providing a means of diversification so that the search can effectively avoid previously encountered local optima (i.e., maximal cliques). Vertex penalties are dynamically updated through the search every $pd$ (penalty delay) iterations. Finally, search stagnation is resolved using a perturbation mechanism that works differently depending on the value of $pd$. If $pd > 1$, i.e., penalties are decreased only occasionally, then the perturbation mechanism reduces the current clique to the last vertex $v$ that was added to it. The rationale is that since the re-
moved vertices all have penalties, they are unlikely to be added back into
the current clique during the next iterative improvement phase. Otherwise, if
\(pd = 1\), penalties are effectively not used at all (since an increase in any ver-
tex penalty is immediately undone), and thus the perturbation mechanism
instead chooses a random vertex \(v \in V\), adds it to the current clique \(K\), and
removes all vertices \(v^* \in K\) that are not adjacent to \(v\). DLS-MC reaches
state-of-the-art performance on many DIMACS benchmark instances and
shows considerable improvement over other approaches such as RLS and
DAGS.

**Phased Local Search**

Shortly after DLS-MC was proposed, the work was further extended into
Phased Local Search (PLS) [15]. PLS, like its predecessor DLS-MC, uses dy-
namic vertex penalties to ensure diversity throughout the search. However,
unlike DLS-MC, PLS uses an adaptive penalty delay parameter, resolving
the issue of instance-specific parameter tuning. In particular, the optimal
penalty delay parameter calculated by DLS-MC was found to be correlated
with the percentage of vertices having \(pd > 0\), and thus PLS, after each
penalty update, adjusts the \(pd\) parameter so that 75% of the total vertices
have \(pd > 0\). PLS achieves state-of-the-art performance on many DIMACS
benchmark instances, and recently discovered a larger maximum clique than
the one previously thought to be optimal for one of the DIMACS benchmark
graphs [66]. PLS was previously applied to the optimal protein structure
alignment problem [29], and in our work is being used for clustering stock
market data. Algorithms 1 and 2 show the PLS pseudo-code.

The set \(C_0(K)\) contains vertices adjacent to all vertices in \(K\), while
\(C_1(K)\) contains vertices adjacent to all but one vertex in \(K\). Thus, the sets
\(C_0(K)\) and \(C_1(K)\) contain vertices corresponding to the iterative improve-
ment and plateau moves, respectively. Generalizing, we can define \(C_p(K)\)
as the set of all vertices not adjacent to exactly \(p\) vertices in \(K\). Formally,

\[C_p(K) = \{i \in V : |K \setminus N(i)| = p\}, \quad p \in \{0,1\}\]

In this definition, \(p\) is restricted to \(0,1\), since any other set \(C_p(K)\) for
\(p > 1\) represents downhill moves, and PLS does not perform any downhill
moves. Instead, the *Initialize* and *Reinitialize* functions represent pertur-
bation mechanisms which help to diversify the search and escape from local
optimum. The *Perturb* function performs either a minor perturbation or a
relatively large perturbation of the current solution. The less destructive
Chapter 3. Related Work

Algorithm 1 The Original (Sequential) PLS Algorithm

<table>
<thead>
<tr>
<th>Algorithm PLS ( (G, tcs, max_selections) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: graph ( G ), integers ( tcs ) (target clique size) and ( max_selections )</td>
</tr>
<tr>
<td>Output: A clique of cardinality ( tcs ) or 'failed'</td>
</tr>
<tr>
<td>( selections \leftarrow 0 )</td>
</tr>
<tr>
<td>( U \leftarrow \emptyset )</td>
</tr>
<tr>
<td>(&lt;\text{Randomly select a vertex } v \in V, K \leftarrow {v} &gt;)</td>
</tr>
<tr>
<td>repeat</td>
</tr>
<tr>
<td>( \text{Phase}(50, \text{RandomSelect}, \text{Reinitialize}) )</td>
</tr>
<tr>
<td>if ( \left</td>
</tr>
<tr>
<td>( \text{Phase}(50, \text{PenaltySelect}, \text{Reinitialize}) )</td>
</tr>
<tr>
<td>if ( \left</td>
</tr>
<tr>
<td>( \text{Phase}(100, \text{DegreeSelect}, \text{Initialize}) )</td>
</tr>
<tr>
<td>if ( \left</td>
</tr>
<tr>
<td>until ( \left( \text{selections} \geq max_selections \right) )</td>
</tr>
<tr>
<td>return 'failed'</td>
</tr>
</tbody>
</table>

Perturbation, also called **Reinitialization**, randomly chooses a vertex \( v \) and adds it to \( K \), then removes all vertices in \( K \) not connected to \( v \) (in order to maintain a proper clique). The more destructive perturbation mechanism, also referred to as **Initialization**, resets the current clique \( K \) to contain only one randomly chosen vertex.

The three phases, also called the **sub-algorithms**, are Random, Degree and Penalty, and are wholly contained within the **Select** and **Perturb** functions. Within the **Select** function, the sub-algorithms choose vertices from \( C_0(K) \) or \( C_1(K) \); Random chooses uniformly at random, Degree chooses a vertex with maximum degree in \( G(V) \), and Penalty chooses the vertex with a minimum vertex penalty. In both the Degree and Penalty sub-algorithm, ties are broken uniformly at random.

Figure 3.1 illustrates an example of the algorithm on a small graph. \( K = \{A,B,C\}, C_0(K) = \{D,E,F\} \) and \( C_1(K) = \{G\} \). If the algorithm is starting a new iteration using the **Degree** sub-algorithm, then it will choose randomly from \( C_0(K) \). If node \( D \) is selected, then \( K = \{A,B,C,D\}, C_0(K) = \{E\} \) and \( C_1(K) = \{F\} \).

We also point out here the ability of PLS to find not only maximum cliques, but also maximal cliques. In fact, in each iteration of PLS (one execution of the outer-most loop in the **Phase** function), at least one maximal clique is discovered. Recall that PLS alternates between phases of iterative improvement and plateau search. During the iterative improvement phase,
Algorithm 2 The Phase Function for PLS

function Phase (iterations, Select, Perturb)

Input: iterations, function Select, function Perturb

Output: K (current maximum clique)

repeat
  repeat
    while (C0(K) \ U \neq \emptyset) do
      v ← Select(C0(K))
      K ← K \cup \{v\}
      selections ← selections + 1
      if (|K| = tcs) then return K
      U ← \emptyset
    endwhile
    if (C1(K) \ U \neq \emptyset) then
      v ← Select (C1(K) \ U)
      K ← [K \cup \{v\}] \{i\}, U ← U \cup \{i\}, where \{i\} ← K \ N(v)
      selections ← selections + 1
    endif
  until (C0(K) = \emptyset and C1(K) \ U = \emptyset)

iterations ← iterations - 1
UpdatePenalties()
Perturb()

until (selections \geq max_selections or iterations \leq 0)

return K
Chapter 3. Related Work

A single maximal clique is greedily expanded by adding vertices that are connected to all others in the current clique. Upon reaching a plateau (when \(C_0(K)\) becomes empty), the current clique is maximal and as such the algorithm has reached a local optimum. The plateau search phase of PLS then commences, and every time a vertex of the current clique is swapped out for other one, a new clique is formed and greedily expanded until it becomes maximal.

3.2 Applications Reducing to the Maximum Clique Problem

We now review a few interesting applications that are reduced to the maximum clique problem. While there are many more than are listed here, these ones in particular are appealing because they could potentially be applied in an online scenario, and would benefit from a parallel online maximum clique algorithm such as the one presented in this thesis.

Bailey et al. present a graph-based data association method to handle batch observations, which is used to assist mobile robot navigation [36]. Batch observations (i.e., from scanning laser, radar, video) detect sets of features simultaneously (or within sufficiently minimal temporal differences) such that the features can be represented using precise relative coordinates. More specifically, the paper uses a maximum clique approach to the Maximum Common Subgraph problem, where each subgraph is a feature graph from a single observation. From each pair of feature graphs, a correspondence graph is generated, and from that the maximum clique is extracted, which represents the maximal set of common features from the two feature graphs.
Chapter 3. Related Work

In a more recent paper [35], which builds on the work by Baily et al. described above, maximum cliques are extracted from correspondence graphs of invariant features, which are then used to solve the well known problem of simultaneous localization and mapping (SLAM). In this approach, a correspondence graph is used whose nodes represent potential matches (i.e., an observation $o_i$ is matched to some landmark $l_j$), and the edges represent the relationship “is consistent with”. The maximum clique in this correspondence graph is the maximum subset of matches which are all consistent with each other, signaling a high probability that a landmark has been identified.

Barnier et al. describe how an air traffic network is converted to a constraint graph, such that the decision variables in the original problem becomes nodes in the graph, and edges connecting nodes indicate one or more constraints have been violated [38]. The basic idea is that these cliques correspond to the most constrained part of the network, and that early instantiations of these variables can result in efficient domain reductions. A simple greedy heuristic is used to find cliques within the constraint graph. From each node in the graph, the algorithm incrementally builds a clique using the node’s adjacency list. Finally, the set of obtained cliques is filtered and only distinct cliques of size greater than two are kept.

With the rise of economical wireless sensor networks, large networks of connected sensors have emerged, requiring new methods to facilitate communication amongst them. Furthermore, minimizing energy consumption is a core challenge since sensors are often deployed in remote areas. To address this problem, Chu et al. have shown how approximate data queries can be used to minimize communication from sensor nodes to base stations, thus minimizing energy consumption [39]. Their approach casts the attributes of the sensor network into a Clique Partitioning problem. An exhaustive and greedy algorithm is then used to extract the cliques from the graph that contain the highest per-attribute data reduction factor (i.e., the data approximation factor). If the sensor nodes are collecting streaming data to be monitored in real-time, then the clique partitioning algorithm must extremely fast and efficient.

3.3 Parallel Metaheuristics

For many years, parallel computing has enabled researchers, in academia as well as industry, to solve many of the most complex problems known to date by harnessing the power of multiple networked computers to accomplish the task in parallel. As described in Section 2.4, SLS methods (and heuristics in
general) have emerged as a practical means of solving complex combinatorial optimization problems. Parallel implementations of heuristic algorithms not only reduce computation time, but can also be used to achieve better solution qualities in the same time as the sequential counterpart. This is a critical feature for large-scale combinatorial optimization systems, especially for those operating in real-time and dynamic environments. Unlike complete, systematic search algorithms, heuristic approaches are inherently well-suited for such dynamic environments. Most well designed randomized heuristics are able to quickly escape from sub-optimal regions of the search space, and have the 'anytime property', meaning they can return the current best solution at any point in time. Furthermore, randomized heuristic algorithms are attractive from a parallelization point of view because independent concurrent runs can, in principle, give better solutions.

In a comprehensive report detailing strategies for the parallel implementation of metaheuristics [67], a basic classification is described, distinguishing between two main approaches: single run and multiple runs. It should be noted here that the terms “single run” and “multiple runs” are used interchangeably with “single walk” and “multiple walks” respectively. In a single run parallelization, the search follows one unique trajectory. Each move through the search space is executed in parallel either by a parallel evaluation of the cost function or by domain decomposition. Multiple runs are further divided into independent and cooperative searches. An independent multiple runs strategy is one in which p search processes start from independently chosen initial points and execute their (potentially different) search strategies independently of one another — no attempt is made to exchange information during the search. When all processes are finished searching, the solutions are gathered (usually by a designated process) and the best overall solution is reported. The other major parallelization strategy, which has gained much attention in recent years, is that of cooperative searching. Many novel metaheuristics have been proposed using a cooperative search framework, and positive results have been reported for a wide range of applications such as bioinformatics [55], network design [68], vehicle routing [51], as well as other fundamental combinatorial problems such as the p-Median problem [69] and the sequential ordering problem [70]. The main advantage of these cooperative approaches is that they are conceptually intuitive, and important aspects of the underlying algorithmic approach can be easily modeled. In general, the consensus is that while parallelization strategies can offer great speed-up potential, cooperative search methods also increase global search robustness [52], allowing for a more thorough exploration of the solution space. While there are many important considerations when
designing a cooperative search strategy, a complete characterization can be achieved by specifying (a) the information which is to be shared, (b) when communications may occur, (c) between which processes information is to be exchanged, and (d) how processes utilize the imported information [71]. Crainic also suggests further classifying parallel cooperative strategies according to whether the information exchanged represents partial or complete solutions [72]. Storing and exchanging partial solutions are typical of adaptive memory strategies which retain high-quality partial configurations in order to avoid unnecessary work such as initial solution construction. Strategies which exchange complete solutions all have a common mechanism by which the solutions are stored and accessed – common names for this type of mechanism include central memory, solution warehouses, pools of solutions or reference sets.

In an initial attempt to rigorously define and characterize parallel metaheuristic search strategies, Crainic et al. propose a comprehensive classification with three dimensions: Search Control Cardinality, Search Differentiation and Search Control and Communication [73]. Although Tabu Search is used as the motivating example, a majority of the results can be generalized to other metaheuristics and stochastic local search methods. Search Control Cardinality examines how the global search is controlled, i.e., either by a single master process or by several master processes, corresponding to 1-control (1C) and p-control (pC) respectively. A typical 1-control strategy would implement a manager-worker architecture, where the manager performs the search and distributes compute intensive tasks (such as the neighbourhood evaluation) to the worker processes. Search Differentiation specifies whether search processes start from the same or different initial starting points, and whether they use the same or different search strategies (even a difference in one or more parameter settings can be considered a different strategy). Four cases are proposed, which are Same Initial Point and Same Search Strategy (SPSS), Same Initial Point and Different Search Strategy (SPDS), Multiple Initial Points and Same Search Strategy (MPSS) and Multiple Initial Points and Different Search Strategies (MPDS). The last dimension of classification corresponds to Search Control and Communications, which define the exchange of information and are encapsulated within the following four classes: Rigid Synchronization (RS), Knowledge Synchronization (KS), Collegial (C) and Knowledge Collegial (KC). Rigid Synchronization describes independent search strategies that do not share any information during the search, and some form of synchronization is usually hard-coded or pre-determined using some parameters. Synchronization in this context refers to portions of code that all processes must execute.
Chapter 3. Related Work

For example, an extreme case is where all processes must reach and execute the same synchronization code before any of the others can continue. Knowledge Synchronization endures the same synchronization constraints as Rigid Synchronization, but allows for some basic communication of information between search processes. The two collegial classes exploit asynchronous communication by using internal logic-driven messaging, thus avoiding hard-coded or pre-determined checkpoints. The main difference between Collegial and Knowledge Collegial is that the former uses basic communication in the sense that the same messages are sent and received, while the latter describes much more complex communication schemes where the messages sent and received are conditioned on a variety of run-time variables.

According to this classification scheme, an independent multiple runs search strategy falls within the pC/RS taxonomy, since each process is in control of its own search trajectory (pC) and no information is exchanged during the search (RS). Cooperative searches, however, belong to the pC/C or pC/KC classes of parallel algorithms, since each process is in control of its own search trajectory (pC) and some amount of information is communicated during the search (C or KC).

3.4 Stochastic Local Search in Dynamic Environments

This thesis considers the online maximum clique problem where edges are dynamically added to or removed from the graph. There is an abundance of literature on dynamic graph problems, but very little which is specific to stochastic local search. In this section we review recent work which addresses the issue of stochastic local search algorithms which can operate in dynamic environments where the underlying problem is constantly changing.

Stochastic Local Search Methods for Dynamic SAT

One of the most prominent problems to which SLS algorithms have been successfully applied is the Satisfiability problem (SAT). While this thesis does not deal specifically with SAT problems, many combinatorial problems can be encoded into SAT and then solved using any SAT solver. This strong correspondence between SAT and other combinatorial problems allows many of techniques used in solving SAT problems to be generalized to other problem domains utilizing SLS algorithms. Hoos and O’Neill were the first to introduce the dynamic SAT problem (DynSAT). They present an
initial investigation that considers several approaches for solving this problem [74]. DynSAT is a generalization of conventional SAT problems which allows for changes in the problem over time.

Four approaches are proposed for dealing with the dynamic SAT problem:

1. Solve a series of SAT instances using existing SAT methods with no alterations. Each new instance is solved by restarting the search when the problem changes. In the case of an SLS algorithm with random initialization, this approach is referred to as random restart.

2. Using an existing SAT algorithm, but rather than restarting the search for each new instance, continue the search from the point in the search trajectory where the change occurred. This is referred to as trajectory continuation.

3. Design a specialized algorithm that tries to locate promising starting points after a change has occurred.

4. Design a specialized algorithm that exploits given or learned knowledge about the dynamics of the problem, i.e., statistical information about the frequency, magnitude and probability of certain changes in order to steer the search towards solutions which tend to be more robust with respect to anticipated future changes.

Their paper only attempts to address approaches 1 and 2, since they require few modification and are not problem specific like the last two approaches. In the paper the authors test their hypothesis on both Random 3-SAT instances and SAT-encoded Graph Colouring instances taken from the SATLIB Benchmark Library [75]. For each test instance used, a 10-stage DynSAT instance was constructed, such that each consecutive stage ensures a different SAT model than the previous stage. The results show that for the Random 3-SAT instances, the search cost using trajectory continuation is approximately a factor of 2 lower than when using random restart, while for the structured Graph Colouring instances it provide little or no benefits. The trajectory continuation approach is enhanced for structured instances by incorporating a soft restart strategy, which essentially performs a random restart when no improvement over the incumbent solution has been made in a given number of search steps. Furthermore, it was found that the improvement seen by using trajectory continuation is orthogonal to the underlying algorithm used, and independent of specific problem instance.
Chapter 3. Related Work

As we will see throughout this thesis, well-designed randomized heuristic mechanisms that make SLS algorithms ideal for hard combinatorial problems, also make them ideal candidate algorithms for time-constrained dynamic environments where the underlying problem is constantly changing.

Online Stochastic and Robust Optimization

The work of Bent and Van Hentenryck considers online stochastic optimization problems where time constraints limit the number of offline computations that can be made in between online processing [76]. The proposed algorithms are general and applicable to many problems, but they are presented in the context of the packet scheduling and multiple vehicle routing (MVR) problems. As in other online problems, the data is assumed not to be available a priori, but rather it is revealed incrementally during algorithm execution. The framework presented assumes that a distribution of future requests is available for sampling. Algorithm 3 depicts the generic online optimization routine described in the paper, and works as follows: given a time horizon $H$, a number of requests $R_t$ available at each $H_t \in H$, and a score $w(r)$ associated with each request, find a feasible solution $\sigma$ maximizing $\sum_{t \in H} w(\sigma(t))$. Offline optimizations are encapsulated in ChooseRequest, and due to time constraints, may only be executed a certain number of times at each step. Furthermore, in their paper, several variants of the online algorithm are given, all of which differ only in the way they implement the ChooseRequest function.

Algorithm 3 Online Optimization

```plaintext
input: time horizon $H = [H_0,H_f]$
$R \leftarrow \emptyset$
$w \leftarrow 0$
for $t \in H$ do
    $R \leftarrow$ AvailableRequests$(R,t) \cup$ NewRequests$(t)$
    $r \leftarrow$ ChooseRequest$(R,t)$
    ServeRequest$(r,t)$
    $w \leftarrow w + w(r)$
    $R \leftarrow R \setminus \{r\}$
```

The first two variants of the online algorithm are Expectation and Hedging. Expectation chooses the action maximizing expectation at each time step by generating future requests by sampling, and evaluating each available request against the sampled requests. If there is enough time for
Chapter 3. Related Work

Offline samples in between each online optimization, then each request is evaluated against $\phi/|A|$ scenarios; this is appropriate when $\phi$ is large enough to make the sampling process meaningful, but if $\phi$ is too small, the algorithms do not yield much information. **Hedging** is an online adaptation of robust optimization, which as the name suggests, attempts to hedge against the worst-case scenario. At each time step, a solution is sought whose deviation with respect to the optimal solution is minimal over all scenarios. Like **Expectation**, the **Hedging** algorithm samples future requests and uses these to choose a request such that the deviation is minimized. As mentioned previously, these two online algorithms are completely encapsulated within the **ChooseRequest** function of the generic online algorithm.

The next two variants, **Consensus** and **Regret**, are used to approximate **Expectation** and **Hedging** when the time in between optimizations is small and there is a large number of requests to serve. They key idea behind these algorithms is that instead of evaluating $\phi/|A|$ scenarios for each available request, $\phi$ scenarios are evaluated by evaluating each sample once against all available and sampled requests. The main benefit is that the samples do not need to be partitioned among available requests, but the limitation is that only one request is chosen to be optimal for each sample while other requests are simply ignored. This is important considering that several requests may be almost identical for a given sample, or that some requests may not be the best overall for a single sample, but may be the best in terms of robustness over all the samples. In other words, the **Consensus** algorithm is very greedy and elitist, while the **Regret** algorithm addresses these issues without performing additional optimizations.

This work demonstrates how online stochastic algorithms can improve the quality and robustness of their solutions if the distribution of incoming data is known. This solution is particularly important for problems where a limited amount of time is available for off-line computation, for example real-time financial analysis where the data arrival rate is often sporadic.

3.5 Online Graph Algorithms for Dynamic Environments

This section presents two recent papers which deal with dynamic graph problems relevant to the one presented in this thesis. The first is particularly relevant as it considers the problem of tracking all maximal cliques in a dynamic graph. The second one is presented as an illustration of a graph-based clustering algorithm which is able to operate in a dynamic environment.
Finding All Maximal Cliques in Dynamic Graphs

Recently, Stix presented an algorithm for tracking all maximal cliques in a dynamic graph [77]. A dynamic graph is represented as a series of graphs $G_0, G_1, \ldots, G_T$ with a constant set of vertices and a changing set of edges. Each graph $G_i = (V, E_i)$, with $0 \leq i \leq T$, corresponds to the graph at time $i$ obtained by thresholding the complete graph using a threshold value of $t$; i.e., $(u, v) \in E_i$ if $\text{distance}(u, v) \leq t$. More specifically, the threshold $t \in [0, \infty]$, thus $0 \leq i \leq T$ represents a discrete set of threshold values, so we can revise the last definition to be $(u, v) \in E_i$ if $\text{distance}(u, v) \leq t_i$, where $t_i$ is the threshold level at time $i$. Increasing the threshold can only remove edges, while decreasing the threshold can only add edges. Thus, because the adding/removing of vertices is monotone with respect to the directional change of the threshold, the dynamic graph problem considered in this paper is an incremental/decremental one. The key advantage to the algorithm by Stix is that the information about cliques obtained for $G_i$ can be used to efficiently determine the new set of maximal cliques in $G_{i+1}$ without having to recalculate the entire problem from scratch. Moreover, knowledge of the complete set of maximal cliques directly implies knowledge of the maximum clique.

Most research thus far in dynamic graph optimization has focused on connectivity and related problems. In these contexts, the graph is partitioned into a crisp set of non-overlapping elements. In this paper, the result over the dynamic graph is a fuzzy clustering representing sets of maximal cliques obtained from the graph constructed by incrementally increasing or decreasing the threshold value. Furthermore, a nice by-product of this approach is that a hierarchical structure evolves as the threshold increases or decreases.

The author provides two algorithms, one for adding edges (decreasing $t$), and one for removing edges (increasing $t$). The problem is formulated as a single step decomposition, meaning that only one edge is added each time. Although it can be expected that more than one edge will be added from $G_i$ to $G_{i+1}$, one can easily construct a new series of graphs such that each one corresponds to adding exactly one edge:

$$|E^*_i \setminus E^*_{i-1}| = 1 \quad i = 1, \ldots, T$$

(3.1)

The above construction can be realized by constructing $k-1$ intermediate graphs whenever $|E^*_i \setminus E^*_{i-1}| = k > 1$.

The following theorems for adding and removing edges are proposed. Supporting lemmas are given in the paper, but are left out here for brevity.
Chapter 3. Related Work

Adding Edges

Let $G_i = (V, E_i)$ be a graph at time $i$, and $(u, v)$ be an edge added from time $i$ to $i + 1$. Furthermore, let $C_i$ be the set of maximal cliques in $G_i$. Then:

1. All cliques in $C_i$ that do not contain either $u$ or $v$ are also in $C_{i+1}$.

2. For all pairs of cliques $(A, B) \in C_i$ such that $u \in A$ and $v \in B$ (or vice-versa), the following statements are true:
   
   (a) $L = (A \cap B) \cup \{u, v\}$ is a clique and $L \in C_{i+1}$ if it is maximal.
   
   (b) $|A \setminus B| = 1 \Rightarrow A \notin C_{i+1}$; otherwise $A \in C_{i+1}$ if it is maximal.

   An analogous statement holds for $|B \setminus A| = 1$.

3. The set $C_{i+1}$ is fully determined by the above statements.

Removing Edges

Let $G_i = (V, E_i)$ be a graph at time $i$, and $(u, v)$ be an edge removed from time $i$ to $i + 1$. Furthermore, let $C_i$ be the set of maximal cliques in $G_i$. Then:

1. All cliques in $C_i$ that do not contain both $u$ and $v$ are also in $C_{i+1}$.

2. For all $A \in C_i$ such that $(u, v) \in A$, the following statements are true:
   
   (a) $A \notin C_{i+1}$
   
   (b) $L = A \setminus \{u\}$ is clique and is in $C_{i+1}$ if it is maximal.

   An analogous statement holds for $L = A \setminus \{v\}$.

3. The set $C_{i+1}$ is fully determined by the above statements.
Statement 1 simply says that a clique is not affected by a removal of an edge if it didn’t contain that edge. Statement 2(a) states that any clique containing the edge \((u, v)\) is destroyed and cannot be in \(C_{t+1}\). Statement 2(b) describes the resulting cliques when \(u\) and \(v\) are removed from consideration.

This work is relevant in the context of this thesis because it introduces the first maximal enumeration algorithm for dynamic graphs. Unfortunately, these theorems and supporting lemmas do not hold for randomized algorithms since no guarantees can be made with respect to the completeness of the set maximal cliques at any given time.

The Star Clustering Algorithm

Aslam et al. present an online graph-theoretic clustering algorithm that uses an approximate clique cover to produce a clustering of the elements (i.e., the vertices of the given graph) [78]. More specifically, their method identifies a set of dense star-shaped subgraphs to cover the vertices of a graph \(G\) thus producing a fuzzy clustering of the elements of \(G\). Each star-shaped subgraph is composed of a star center and multiple satellite vertices, where an edge exists between each satellite and the star-center, but not necessarily between satellite vertices. Figure 3.2 (A) shows a star cover for a small set of vertices, where larger circles are the star centers and smaller circles are the satellites adjacent to each star center. This approach guarantees pairwise similarities between satellites and the star-centers, while only a lower-bound and expected similarity between satellite vertices is given. The

![Figure 3.2: A star cover before (A) and after (B) a new vertex is added.](image)

authors present both offline and online algorithms for producing a clustering by star covers of a given graph. The online algorithm assumes a cluster exists and new vertices arrive incrementally over time (insertion), and also that existing vertices can become obsolete (deletion). No consideration is made for edge removals or additions with respect to existing vertices. When a
new vertex $v$ is added to the graph, the degree of that vertex determines how the existing clusters will be affected. If $v$ does not have an edge to any existing star center, then it becomes a star center. If $v$ is connected to any other vertex $u$ such that $\text{degree}(u) \geq \text{degree}(v)$, then $v$ becomes a satellite of $u$. These are the simple cases; the difficult cases are (1) the degree of $v$ is greater than all other star centers it is connected to, and (2) adding $v$ increases the degree of an adjacent satellite such that it has a higher degree than its associated star center. In these cases, the existing star cover is broken and must be re-calculated. All affected vertices (i.e., those adjacent to $v$ or in the broken star cover) are enqueued into a list so that the star cover can be re-calculated. The star cover reconstruction process is started using the vertices in this list. These local changes may further break other star covers, causing more vertices to be added to the list. This entire process is repeated until all affected vertices have been processed and form a new stable star cover. Figure 3.2 shows the star cover before (A) and after (B) the addition of a single vertex which results in the current star cover being broken and re-organized.

Unlike the algorithm described in 3.5, where the number of vertices remain constant and only edges are added or removed, the star clustering algorithm demonstrates how a clustering of the graph vertices can be maintained when new vertices are added or removed from the graph. Furthermore, similar to the other approaches described in this section, the algorithm is able to repair the set of clusters (star covers) without re-starting from scratch.

3.6 Computational and High Frequency Finance

The work we present in this thesis is partially motivated by several other projects that have demonstrated the ability to extract meaningful information from stock market data. Stock market data can be roughly categorized into ‘historical’ and ‘real-time’ data. Until recently, historical data has been available and more easily accessible than real-time data. Like many other businesses, financial institutions consider their data as a competitive edge and rarely provide this data to the public, or even academic institutions for that matter. Fortunately, recent technological movements in the financial industry have resulted in stock markets moving to either partially or completely electronic-based systems. For example, the NASDAQ is a completely electronic exchange that not only distributes the trade and quote data in real-time, but also accepts new trade and quote orders electronically. The following describes several research projects which motivate our approach.
Boginski et al. use historical prices to reveal complex price-price interaction networks [13]. Using these networks, they are able to find clusters of stocks which have highly correlated price trajectories over a long time interval. In a similar study, the stock market is shown to be a nearly-decomposable network which comprises hierarchic subsystems [34]. This approach also uses historical data, but emphasizes the ability to further classify stocks into multiple levels using the price-price interaction patterns.

Another interesting project is by Nesbitt and Barrass who attempt to find trading patterns using a combination of visual and auditory displays [79]. In this project, day traders can visualize real-time quote data using a novel bid-ask-landscape, complemented with auditory cues, in order to accurately predict short-term price movements. While this approach emphasizes a user-driven trading system, the authors suggest that through supervised training models, heuristics could potentially be extracted from the trader and used in an automated trading environment.

Michael Kearns et al. have developed the Penn-Lehman Automated Trading Project (PLAT) [80], which is a state-of-the-art simulation platform for developing and testing automated trading agents. Each trading agent employs a particular trading strategy, and aims to maximize profits for a single stock over a single trading day. All trading strategies attempt to completely unwind share positions before the end of the trading day. Dempster et al. have produced a series of interesting research papers that explore the profitability of real-time technical analysis trading in the foreign exchange (FX) market [81, 82, 83, 84]. The main conclusions of their work was that using single technical indicators in isolation ultimately results in a loss-making strategy [82]. Further research by the group, however, showed that by combining multiple technical indicators and strategies using genetic programming, they were able to produce a statistically significant profit-making strategy returning 7% per annum [83].
Chapter 4

Parallel Dynamic Local Search and Phased Local Search

Even when intelligent heuristics are designed to efficiently find the maximal clique in a graph, the inherent complexity of the problem makes it challenging to produce accurate results within reasonable times. While researchers continue to produce new algorithms for tackling these difficult problems, the most promising approach for cutting down the overall computation time required to find high-quality solutions within tight time constraints is parallelization. Parallel SLS algorithms provide a natural way to design and implement efficient and robust algorithms for solving hard combinatorial optimization problems. By robust, we mean the algorithms offer a consistently high level of performance over a variety of problem instances, and efficiency means that it can solve large problems within a reasonable time — where “reasonable” depends on the context under consideration; e.g., an offline problem using a massive data set versus an online/real-time problem with a smaller data set but more stringent time constraints. Thus, parallel SLS is an attractive approach for achieving high-quality solutions in time-constrained systems, and in particular, when there is a trade-off between compute time and solution quality. The focus of this chapter is on the design, implementation and empirical analysis of parallel versions of two new state-of-the-art algorithms for the maximum clique problem: Dynamic Local Search (DLS-MC) [64] and Phased Local Search (PLS) [15]. We show the speedup obtained through multiple independent runs, a very basic yet highly efficient parallelization strategy. For both implementations we provide a detailed empirical analysis on their scalability up to 32 processors using a small but representative set of DIMACS benchmark instances.
4.1 Independent Multiple Runs

A multiple independent runs parallelization strategy is essentially launching \( p \) independent runs of the algorithm in parallel, where "independent" refers to the fact that individual search processes do not share information during the search (and hence do not act upon any external information to bias their own search trajectory). To be completely independent, search processes must also perform independent initialization, ruling any out possibility of correlation in the search trajectory. We take a slightly different approach to this strategy in that we perform a controlled initialization where the manager process seeds each worker process with a unique starting vertex.\(^4\) Although one could argue that the runs are not completely independent, the randomness involved in DLS-MC and PLS (and for any SLS algorithm in general) will cause the search trajectories to become quickly uncorrelated. In Section 4.4.1 we perform a small test to show how our modified approach to multiple independent runs affects the overall speedup results. For simplicity, through this chapter and the remainder of the thesis, we refer to our modified scheme as multiple independent runs.

4.1.1 Overview

This section discusses the design and implementation of Parallel Phased Local Search (PPLS) and Parallel Dynamic Local Search (PDLS-MC), parallel implementations conforming to the multiple independent runs strategy. In our implementation, a manager-worker architecture is used to control the communication that is needed to exchange and report on solutions obtained after each run. We point out here that a "run" refers to a single search trajectory that terminates when either the target clique size has been found, or a user-specified limit on the run-time (either in terms of CPU time or search steps) has been reached. Unless otherwise noted, our experiments use the known optimal clique size as the target clique size. In our manager-worker architecture there is one manager and many worker processes, and the manager process does not perform any searching. For our experiments, each process is mapped to a single processor, although this is not strictly necessary to obtain empirical speedup results, since we can measure the run-length and convert it to run-time (see Section 4.1.3 for details). For example, "Parallel PLS with eight search processes" means we have one manager pro-

\(^4\)In the case where there are more workers than vertices, some workers will be assigned the same starting vertex.
cess, eight worker processes, executing on nine different processors.\footnote{Some of our machines have dual CPU's, in which case two processes may be mapped to the same machine, but they are executing on different processors.}

Because no information is exchanged between the individual runs, the only work performed by the manager is at the beginning and end of each run. At the start, the manager seeds the workers with a unique starting vertex, and then waits for a worker to indicate it has quit searching. If a worker quits searching because it found the target clique size, it sends a ‘SOLUTION.FOUND’ message to the manager, who then relays this message to the other workers; otherwise if the manager receives a ‘TIME.EXCEEDED’ message, it does not need to pass on this message, since this simply indicates that the worker who sent the message has exceeded its maximum run time or number of search steps. When a worker receives the message indicating that another worker has found the target clique size, it exits the search routine immediately. The manager then gathers the best clique sizes from all the workers, determines the largest size and the set of all cliques of that size (since more than one maximum clique may have been found), and then proceeds by requesting the actual solutions (vertices in the maximum clique) from the appropriate worker(s). This exposes a large-grained parallelization architecture as the only communication performed during the run is non-blocking asynchronous message-checking, and synchronization between the worker processes is only performed at the beginning and the end of each run. Figure 4.1 shows a flowchart of the multiple independent runs parallelization strategy. As outlined in Section 3.3, parallel SLS methods can be categorized according to several criteria. Following the proposed notation, our particular implementations conform to a p-Control, Rigid Synchronization, and Multiple Point Single Strategy (pC/RS/MPSS) parallelization scheme: all processes control their own search, no information is exchanged and/or used during the search, and processes execute the same search algorithm starting from different initial points.

4.1.2 Implementation Considerations

Upon finding a solution, the worker process must notify the others so that they can terminate their search. In order to implement this, worker processes must occasionally check for the ‘SOLUTION.FOUND’ message. The interval at which to check for this message is a user-defined parameter called poll\_interval. In general, most SLS algorithms are designed such that they rely on very efficient and fast iteration cycles, and checking for these messages every iteration would certainly cause the performance of the algorithm...
to degrade considerably. Conversely, if no polling mechanism exists, then when one process finds the target clique size, all others will continue searching until either they too find the target clique size, or the maximum number of iterations is exceeded. Clearly, this is a waste of valuable computing resources, both in practice and within a testing environment. Instead, we simply choose to poll in regular intervals throughout the search (i.e., every \( k \) steps for some parameter value \( k \)) such that performance is not hindered, yet provides a reasonable upper bound on the duration between successive message checks. More details on this specific issue are discussed in Section 4.3.

Another topic worthy of discussion here is that of accurate timing. The first round of experiments was timed using MPI's MPI_Wtime() method call, which measures wall clock time. We soon found out that this approach is prone to being inaccurate if the machine load cannot be precisely controlled. Furthermore, for hard instances which required considerable compute time, the variable containing the elapsed time in milliseconds was often subject to an overflow error. To remedy this problem, we measured the exact number of CPU cycles using assembly code to directly access the CPU counter register \([85]\). We then extracted the executing machine's exact CPU speed from /proc/cpuinfo (the "cpu MHz" line) in order to obtain a precise measurement of elapsed run-time.
4.1.3 Cost Model

Due to the inherent added complexity when parallelizing a sequential algorithm, we provide a basic cost model describing the algorithm's run-time that incorporates the communication overhead, enabling us to estimate the portion of run-time spent on communication versus searching. Furthermore, the cost model allows us to calculate the run-time of the algorithm by measuring the run-length, which is important for an empirical analysis if dedicated machine resources are hard to obtain. In our implementation of an multiple independent runs algorithm, a worker process only communicates with the manager to check for a termination condition (i.e., to see if another worker has found the target clique size). We define the run-time cost model as:

\[
RT = (\text{selections} \times T_s) + (\text{num.polls} \times T_p) \tag{4.1}
\]

where \(\text{selections}\) is the number of selections made during the execution of the algorithm, \(\text{num.polls}\) is the number of times the process checks for incoming messages, \(T_p\) is the time (in CPU seconds) per poll, and \(T_s\) is the time (in CPU seconds) per selection, which depends on both the algorithm and the problem instance on which it is measured.

The \(\text{num.polls}\) variable is dependent on a user-defined parameter \(\text{poll.interval}\), which specifies how often to poll for messages; thus we can easily calculate \(\text{num.polls}\) as follows:

\[
\text{num.polls} = \left\lfloor \frac{\text{selections}}{\text{poll.interval}} \right\rfloor
\]

We can then rewrite Equation 4.1 as

\[
RT = (\text{selections} \times T_s) + \left( \frac{\text{selections}}{\text{poll.interval}} \times T_p \right)
\]

Rearranging to solve for \(T_p\), we get:

\[
T_p = \frac{RT - (\text{selections} \times T_s)}{\frac{\text{selections}}{\text{poll.interval}}} \tag{4.2}
\]

In order to determine \(T_s\) or \(RT\), we need a second equation. Fortunately, we can approximate the value of \(T_s\) by calculating the CPU time for a single vertex selection when the algorithm runs without message polling (this is essentially equivalent to the corresponding sequential algorithm, except with some minor differences in the startup and shutdown cost factors such as MPI.)
initialization and seeding of unique starting vertices). We then obtain this simplified run-time model:

\[ RT^* = (\text{selections} \times T_5^*) \]

and rearranging for \( T_5^* \) we get

\[ T_5^* = \frac{RT^*}{\text{selections}} \quad (4.3) \]

We calculated \( T_5^* \) by running 100 independent runs on a controlled machine, with no other user processes competing for CPU cycles. Our reference machine contains a single Intel Pentium 4, 2.80GHz processor CPU with 512KB cache, 512MB of RAM, running SuSe Linux with kernel version 2.6.16.13-4-smp.

For each run we specified a target clique size larger than existed in the graph, forcing the algorithm to perform an exact number of selections, and thus ensuring that \( T_s \) was truly representative of the average time per selection, taking into account that the algorithm executes different parts of the code depending on the current state of the search. For example, when doing plateau search, a vertex \( v \) is taken from the set \( C_1(K) \) and swapped with a vertex \( v^* \) in the current clique \( K \) if \( (v, v^*) \notin E \). DLS-MC and PLS choose between two options for finding \( v^* \): look through vertices in \( K \) for one not connected to \( v \), or traverse \( \overline{L}_v \) (\( v \)'s complementary adjacency list) until \( v^* \) is found. Clearly, both of these operations are linear in the size of \( K \) and \( \overline{L}_v \), respectively, but for efficiency reasons choosing to search for \( v^* \) in the smaller of the two is desirable. This run-time decision causes slight variations in time required for a single selection, and so by running the algorithm for a large number of selections, we are able to record an average over all selections representing a reasonable approximation for \( T_s \) (see Table 4.1).

To calculate \( T_p \), we used the same reference machine and test setup as used for calculating \( T_5^* \). The only difference is that we used the PPLS algorithm with polling, setting \textit{poll.interval} = 10 (selections). The reason for choosing this value, which may seem quite low, is twofold. First, we felt that it was important to verify how such frequent polling impacted the algorithm, because when used in an online setting, it must have a fast response time. Second, this value is a reasonable balance between polling too often and degrading the performance of the algorithm, and polling too little, resulting in a waste of value computing resources. Refer to Table 4.1 for full results. A more practical solution to this problem of determining an optimal polling interval is addressed in Section 6.4.4 in the context of an
Chapter 4. Parallel Dynamic Local Search and Phased Local Search

Table 4.1: Values for $T_s$ and $T_p$ determined on our reference machine on each of our four benchmark instances; see text for details.

<table>
<thead>
<tr>
<th>Instance</th>
<th>PLS</th>
<th>DLS-MC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_s^*$ [sec]</td>
<td>$T_p$ [sec]</td>
</tr>
<tr>
<td>brock800.1</td>
<td>$4.322 \times 10^{-6}$</td>
<td>$1.565 \times 10^{-6}$</td>
</tr>
<tr>
<td>keller6</td>
<td>$9.454 \times 10^{-6}$</td>
<td>$3.134 \times 10^{-6}$</td>
</tr>
<tr>
<td>C1000.9</td>
<td>$2.174 \times 10^{-6}$</td>
<td>$2.150 \times 10^{-6}$</td>
</tr>
<tr>
<td>p_hat1500-1</td>
<td>$1.458 \times 10^{-6}$</td>
<td>$2.524 \times 10^{-6}$</td>
</tr>
</tbody>
</table>

Table 4.2: Estimated run-time (Est), actual run-time (Act), and estimation error (Err) for DLS-MC and PLS on our four benchmark instances using the cost model defined in Equation 4.1.3.

<table>
<thead>
<tr>
<th>Instance</th>
<th>DLS-MC</th>
<th>PLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>brock800.1</td>
<td>97.4</td>
<td>92.9</td>
</tr>
<tr>
<td>keller6</td>
<td>248.9</td>
<td>231.8</td>
</tr>
<tr>
<td>C1000.9</td>
<td>52.7</td>
<td>48.9</td>
</tr>
<tr>
<td>p_hat1500-1</td>
<td>291.9</td>
<td>292.3</td>
</tr>
</tbody>
</table>

The estimate for the p_hat1500-1 was very close for both algorithms,
while the estimation error for the brock800_1 and C1000.9 instances was off for both DLS-MC and PLS. The keller6 estimate was close for PLS but not for DLS-MC (0.1% and 7.4% respectively). One possibility points to the small run-time values, especially for C1000.9 and p.hat1500-1, which invite a greater chance of timing errors. Another possibility is a bad estimate for the value of $T_p$, which would in turn affect the estimated run-time. Since performing a message probe using MPI makes a series of system calls, it is possible that subtle low-level system events such context switching or page faults could have affected the measurement of $T_p$. Finally, considering the granularity of our timing measurements, even small rounding errors in the calculations could be to blame. In retrospect, it would have been better to devise an experimental approach which was less sensitive to precise timing measurements. We also note that the difficulties endured in these timing experiments highlight the fact that the excellent performance of SLS algorithms stems from the fact that many cheap steps can be performed within relatively short periods of CPU times.

The final step in presenting the cost model is to verify that the parallelization does not affect the relationship between search selections and run-time. Figure 4.2 shows two scatter plots for PPLS illustrating how the number of selections is correlated with the run-time. Each point on the plot represents the time needed to execute the corresponding number of selections. For the case of a sequential algorithm, each run of the algorithm produces a single point on a scatter plot. Similarly, for the parallel case, each run with $p$ processors results in $p$ points on the scatter plot. These results confirm there is a linear relationship between the number of selections and run-time for both the sequential and multiple independent runs versions.

### 4.2 Cooperative Search Strategy

This section explores the effectiveness of a cooperative search parallelization strategy for PLS. The resulting algorithm, called Cooperative Parallel Phased Local Search (CPPLS) is similar to the non-cooperative variant introduced earlier in that the communication model follows a manager-worker architecture. The difference now is that worker processes are able to communicate with the manager during the search to participate in a solution exchange mechanism. Following the strategy used in several recent cooperative search methods [51, 52, 68, 70], we allow worker processes to share their incumbent solutions (current working clique) during the search. This
Chapter 4. Parallel Dynamic Local Search and Phased Local Search

Figure 4.2: Two sample scatter plots for PPLS, showing the correlation between run-length (x-axis) and run-time (y-axis).

concept of sharing elite solutions is a popular strategy with many similar names: central memory, solution pools, solution warehouse or elite solution set. The intuitive aim of such elite solution sharing is to bias the search to
look in more promising regions of the solution space [70].

**Cooperative Parallel Phased Local Search (CPPLS)**

Referring to the proposed notation for parallelization classification (discussed in Section 3.3), our particular implementation conforms to a p-Control, Collegial, and Multiple Point Single Strategy (pC/C/MPSS) approach: all processes control their own search, information is exchanged and used during the search but the type of information is limited to elite solutions, and processes execute the same search algorithm starting from different initial points.

Previous studies have shown that cooperation through asynchronous exchanges of information is superior to synchronous approaches [73]. We therefore use a strictly asynchronous communication framework to allow for maximum flexibility between the individual processes. In our framework, worker processes send their solution to the manager after every non-successive improvement; e.g., all improvements except during the initial greedy iterative improvement phase where the current solution is improved after every selection. The rational here is that the search has then found an exit of a plateau, and sharing the incumbent solution then allows other search processes to “jump” over to the same area, but with a perturbed landscape due to the different vertex penalties.

The exchange mechanism works as follows: process A sends the size of its incumbent solution to the manager, who then determines if it is better than the worst of the elite solutions. The manager then sends a boolean response indicating whether it wants the solution; if so, process A sends the actual solution (vertices of the clique), otherwise, the manager rejects the incoming solution and the exchange process is aborted. This approach was evaluated using a parallel cooperative GRASP with path-relinking heuristic for the 2-path network design problem [86], and was found to be superior to always sending the solution to the manager without first checking its quality relative to the other elite solutions. The manager stores all the elite solutions in memory and distributes to the workers when requested. A solution is chosen uniformly at random from the set of all elite solutions. The number of elite solutions to store at any given time is currently a user-defined parameter to the algorithm. Too many elite solutions means that many sub-optimal solutions will be collected, while too few elite solutions results in poor diversification since the algorithm will explore the same regions of the search space. A basic empirical evaluation was conducted to determine an optimal value of 10 for our given subset of problem instances. See Section 4.4.2 for
Chapter 4. Parallel Dynamic Local Search and Phased Local Search

details.

Workers processes request an elite solution from the manager when they feel that the search has stagnated. In particular, we have adopted the approach used by Battiti et al. in the Reactive Local Search (RLS) algorithm [58] to detect stagnation: after a specified number of selections since the last improvement or restart, the search is restarted. We employ the same stagnation detection mechanism expect that instead of restarting, a worker requests an elite solution and continues the search using the newly imported solution as its working clique. A solution is imported when the condition selection - last_improved > STAGDELAY is true, where selection is the current selection (steps) of the algorithm, last_improved indicates when the incumbent solution was last improved, and STAGDELAY is a parameter which specifies how many selections without improvement are considered indicative of search stagnation. We use the same calculation for STAGDELAY as used in the RLS algorithm, which is STAGDELAY = \beta \times |K|, where \beta is a user-defined parameter and K is the size of best solution obtained so far. Including |K| into the calculation ensures that the stagnation threshold increases relative to the maximum clique size.

Another interesting approach that we did not explore, although it may be of interest for future work, is the ability to share vertex penalties associated with the current clique. The idea here is that since vertex penalties are associated with recently selected vertices, we can explicitly control the convergence to, or divergence from, certain areas of the solution space. For example, if process A finds a high-quality solution and we want processes B to explore that area, then we could adjust B's vertex penalties to increase the chances that the region found by process A is explored in the near future. Similarly, we could adjust the vertex penalties to force the search to diverge from a specific region.

4.3 Empirical Analysis

This section describes the experimental framework used to empirically analyze the performance of PDLS-MC, PPLS and CPPLS. The main questions we wish to answer are which of these algorithms performs best, and in particular, whether the cooperative approach outperforms the (much simpler) multiple independent runs method. Furthermore, we wish to examine to what extent these parallel implementations scale as additional processors are used. Our evaluation consists of measuring the run-time and run-length distributions on various benchmark instances for PDLS-MC and PPLS, and
comparing the results against their respective sequential versions. We also measure the speedup obtained by executing an increasing number of independent runs in parallel. For CPPLS, we do the same and also measure the speedup relative to PPLS in order to determine how well the cooperative version performs relative to the non-cooperative variant.

The small subset of benchmark instances used in this evaluation are C1000.9, brock800.1, p_hat1500-1 and keller6. The C1000.9 graph is from the random graph family and originally created for graph colouring problems [87]; an instance of the form Cx.y has x nodes and 0.y edge probability. The brock800.1 instance is from the Brockington-Culberson graph family, a set of benchmark instances originally designed to be hard Maximum Independent Set problems by use of “camouflaging” [59]. Both the C1000.9 and brock800.1 instances were chosen because they were the subjects of thorough empirical analysis in the original DLS-MC paper, and thus detailed performance results for the sequential version were already available. The p_hat family of benchmark graphs are generalizations of the classical uniform random graph generator, having a wider node degree spread and larger cliques than uniform graphs [88]. The keller graph instances are graph-theoretic formulations of Keller's conjecture [60], and are generally hard for maximum clique solvers, including DLS-MC and PLS.

Table 4.3 shows the details for each of these instances, with ω representing the best known clique size.

<table>
<thead>
<tr>
<th>Instance</th>
<th># Vertices</th>
<th># Edges</th>
<th>ω</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1000.9</td>
<td>1,000</td>
<td>450,079</td>
<td>≥ 68</td>
</tr>
<tr>
<td>p_hat1500-1</td>
<td>1,500</td>
<td>284,923</td>
<td>≥ 12</td>
</tr>
<tr>
<td>brock800.1</td>
<td>800</td>
<td>207,505</td>
<td>23</td>
</tr>
<tr>
<td>keller6</td>
<td>3,361</td>
<td>4,619,898</td>
<td>≥ 59</td>
</tr>
</tbody>
</table>

Table 4.3: Properties of our selected DIMACS benchmark instances.

For each algorithm and benchmark instance, 100 independent runs were performed. No search time limit was given, however a large selection limit of 200 million was given to allow as many runs as possible to find the target clique size. In all experiments for each instance, with the exception of the keller6 instance, both PDLS-MC and PPLS found the optimal clique size in every run. For the keller6 instance, PDLS-MC always found the target clique size within the given selection limit, but PPLS fell short, only finding the target clique size in 62/100 runs when using one processor and in 86/100 runs when using two processors.
Chapter 4. Parallel Dynamic Local Search and Phased Local Search

There are two general approaches for empirically measuring the performance of SLS algorithms. The more common method is to measure the actual wall-clock execution time of the algorithm over a range of benchmark instances. The other method measures the performance with respect to the number of elementary operations (e.g., selections or steps), which in the context of DLS-MC and PLS refers to a single vertex addition or removal. These two approaches can always be converted from one to the other using a cost model that specifies the “time per elementary operation” (see Section 4.1.3 for details). Empirically measuring the run-time performance of an algorithm is relatively straightforward for non-randomized algorithms. For SLS algorithms, the idea is to determine the probability that the algorithm will find the optimal solution within a given time $t$. If $RT$ is the time it takes for an algorithm to find the optimal solution, then the solution probabilities, $P_s(\text{RT} \leq t)$, can be empirically measured yielding the Run-time Distribution (RTD). The following definition is taken from Chapter Empirical Analysis of the book by Hoos and Stützle [46]:

**Definition:** Consider a Las Vegas algorithm $A$ for decision problems class $\Pi$, and let $P_s(\text{RT}_{A,\pi} \leq t)$ denote the probability that $A$ finds a solution for a soluble instance $\pi \in \Pi$ in time less than or equal to $t$. The run-time distribution (RTD) of $A$ on $\pi$ is the probability distribution of the random variable $\text{RT}_{A,\pi}$, which is characterized by the run-time distribution function $\text{rtd}: \mathbb{R}^+ \mapsto [0, 1]$ defined as $\text{rtd}(t) = P_s(\text{RT}_{A,\pi} \leq t)$.

The definition of a Run-length Distribution (RLD) differs from that of an RTD only in that run-time is measured in terms of the number of steps/selections taken to find the optimal solution. As we mentioned previously, RLDs and RTDs can be converted from one to another by knowing the precise timing of the underlying elementary operations.

Figure 4.3 shows the RLD for DLS-MC on the C1000.9 instance using 16 processors (a), and the two corresponding RTDs (b), one of which was calculated using the measured wall-clock time, while the other was calculated using the cost model as described in Section 4.1.3. These plots show how the cost model can use the run-length to generate a reasonably good estimate of the true run-time. When using the multiple independent runs parallelization strategy, if the run-time (or run-length) of the underlying sequential algorithm is exponentially distributed, then a linear speedup can be achieved [46]. This is because of a well-known statistical result which states that if the probability of finding a solution in time at least $t$ is exponentially distributed with median $m$, then the probability of finding the same solution
Figure 4.3: RLD and RTDs for the C1000.9 instance using 16 search processes.
or better using \( p \) processes is distributed with median \( \frac{m}{p} \) [89]. This means that success probability of the algorithm when run for \( t \) time units is the same as the success probability when running the algorithm \( p \) times for \( \frac{t}{p} \) time units each. Based on these observations, if the RTD of the sequential algorithm is exponentially distributed, then it follows theoretically that we should get optimal parallelization speedup with respect to the number of processors. However, in practice SLS algorithms do not have perfectly exponential RTDs and thus the observed speedup is not always optimal.

To measure the actual parallel speedup of the multiple independent runs strategy, we perform the search using a varying number of processors and record the number of selections (run-length) needed to find the target clique size (recall that any run-length distribution can be converted to a run-time distribution).

If \( RT_{A_S} \) is the CPU time required by the sequential algorithm \( A_S \), and \( RT_{A_p} \) is the time taken by \( A_p \), a parallel version of \( A_S \) using \( p \) processors, then the relative speedup \( S_{Ap} \) of the parallel algorithm is defined as

\[
S_{Ap} = \frac{RT_{A_S}}{RT_{A_p}}
\]

Similarly, we can substitute run-time \( RT \) for run-length \( RL \) and get an equivalent definition for speedup:

\[
S_{Ap} = \frac{RL_{A_S}}{RL_{A_p}} \quad (4.4)
\]

The efficiency of a parallel algorithm describes the average fraction of time that the processors are being effectively utilized. The formula for measuring the efficiency \( \eta_{Ap} \) is given by:

\[
\eta_{Ap} = \frac{S_{Ap}}{p} \quad (4.5)
\]

Clearly, the maximal efficiency of 1 occurs when \( S_{Ap} = p \), meaning that the speedup is linear with respect to the number of processors.

Environment

The underlying distributed communications utilizes the Message Passing Interface (MPI), the de-facto standard message passing library for high performance parallel computing. In particular, we use LAM/MPI version 7.1.1,
which is a high-quality, open-source implementation of MPI [6]. MPI supports multiple platforms in a variety of heterogeneous computing environments, providing a solid basis for designing and implementing scalable parallel architectures for use in a variety of contexts from single multi-processor machines to arbitrary-sized compute clusters. Our primary testing environment consisted of a cluster of 14 PC workstations each equipped with dual Intel Xeon 3.06GHz CPUs with 2GB of RAM, running SuSe Linux 9.1 with kernel version 2.6.5-7.252-smp. Some experiments required more than 28 processors; in these cases, spare (idle) machines were used, and only the RLDs were measured to avoid problems with varying CPU speeds.

4.4 Results

In this section we describe results obtained from the empirical analysis of PDLS-MC, PPLS and CPPLS. In order to keep this section reasonably small and easy to read, we present only sample and illustrative results. The full set of results using all four of our benchmark instances can be found in Appendix A.

4.4.1 PPLS and PDLS-MC

As mentioned at the start of this chapter, we did not implement a true multiple independent runs parallelization strategy. The minor difference is that processes do not begin the search with an independently chosen seed vertex. Instead, the manager assigns a unique seed vertex to each worker, thus ensuring a unique starting point for each search trajectory (unless of course there are more workers than vertices, in which case some trajectories may start from the same seed vertex). Table 4.4 shows the performance difference between a multiple independent runs strategy with and without unique seed initialization. The experiment was performed using 200 independent runs per instance using 13 search processes per run, with a maximum search selection limit of 100,000,000. The target clique size was achieved in all runs for each instance, except for one failed run on keller6 when not using unique seed vertices. As we expected, the results show that using unique seed vertices improves performance on some instances, but also degrades it for others, and that most likely the overall effect is dependent on the problem instance. Further experiments on a wider range of instances would allow us to more thoroughly characterize the performance effect when using unique seed vertices.
Chapter 4. Parallel Dynamic Local Search and Phased Local Search

Table 4.4: Observed speedup when starting with unique seed vertices using 13 search processes.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Median Run-length [selections]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Without unique seeding</td>
<td>With unique seeding</td>
</tr>
<tr>
<td>C1000.9</td>
<td>68,457</td>
<td>44,815</td>
</tr>
<tr>
<td>brock800_1</td>
<td>433,510</td>
<td>442,511</td>
</tr>
<tr>
<td>p_hat1500-1</td>
<td>20,701</td>
<td>9,182</td>
</tr>
<tr>
<td>keller6</td>
<td>10,340,507</td>
<td>11,367,093</td>
</tr>
</tbody>
</table>

Table 4.5 shows the raw numbers for the speedup and efficiency on our four benchmark instances for PDLS-MC and PPLS in terms median run-length.

Figures 4.4 and 4.5 show sample scalability and speedup results for PPLS and PDLS-MC respectively. These plots show the scalability and speedup results for only the brock800_1 instance, although we note that the results were for the most part similar across the different test instances. As mentioned previously, the full set of results for the benchmark instances can be found in Appendix A.

Figure 4.6 contains two related RLD plots for PDLS-MC with 1 and 4 search processes on the keller6 instance. Each plot contains the empirically measured RLD, along with an approximated RLD with an exponential distribution function $ed[m](x) = 1 - 2^{-x/m}$, where $m$ is the median run-length from the RLD.6

Discussion

In order to accommodate a wide range of application scenarios, SLS algorithms must maximize their parallelization speedup; in other words, if we perform a search using $p$ processes, we want to achieve a speedup of $p$ (assuming a one-to-one mapping between processes and processors). We previously defined efficiency as the ratio between speedup and $p$; in other words, efficiency can be thought of as the fraction of time that the processors are being effectively utilized, which is especially important in industry where effective resource management is a key component to any computing infrastructure. Our results indicated near-linear speedup for 2 and 4 processors, but reduced slightly thereafter, with an average efficiency of approximately 0.5 for the 32 processor tests.

6According to Pullan and Hoos [64], the run-time of DLS-MC tends to be exponentially distributed.
Chapter 4. Parallel Dynamic Local Search and Phased Local Search

<table>
<thead>
<tr>
<th>Instance</th>
<th># Proc</th>
<th>DLS-MC Speedup</th>
<th>Efficiency</th>
<th>PPLS Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>brooks800-1</td>
<td>1</td>
<td>1.000</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.038</td>
<td>1.019</td>
<td>2.038</td>
<td>1.019</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.367</td>
<td>0.842</td>
<td>3.367</td>
<td>0.842</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>5.591</td>
<td>0.699</td>
<td>5.591</td>
<td>0.699</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>11.402</td>
<td>0.713</td>
<td>11.402</td>
<td>0.713</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>12.900</td>
<td>0.403</td>
<td>13.602</td>
<td>0.425</td>
</tr>
<tr>
<td>phat1500-1</td>
<td>1</td>
<td>1.000</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.629</td>
<td>1.315</td>
<td>2.629</td>
<td>1.315</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.792</td>
<td>0.698</td>
<td>2.792</td>
<td>0.698</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>5.218</td>
<td>0.652</td>
<td>5.218</td>
<td>0.652</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>13.735</td>
<td>0.858</td>
<td>13.736</td>
<td>0.859</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>18.231</td>
<td>0.600</td>
<td>22.002</td>
<td>0.688</td>
</tr>
<tr>
<td>C1000.9</td>
<td>1</td>
<td>1.000</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.682</td>
<td>0.841</td>
<td>1.682</td>
<td>0.841</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.253</td>
<td>0.813</td>
<td>3.253</td>
<td>0.813</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>6.854</td>
<td>0.857</td>
<td>6.854</td>
<td>0.857</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>11.285</td>
<td>0.705</td>
<td>11.285</td>
<td>0.705</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>16.756</td>
<td>0.524</td>
<td>18.534</td>
<td>0.579</td>
</tr>
<tr>
<td>heller6</td>
<td>1</td>
<td>1.000</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.572</td>
<td>1.264</td>
<td>1.39</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5.120</td>
<td>1.280</td>
<td>2.47</td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7.768</td>
<td>0.971</td>
<td>4.57</td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>14.244</td>
<td>0.890</td>
<td>10.32</td>
<td>0.65</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>28.733</td>
<td>0.900</td>
<td>16.04</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Table 4.5: Speedup and Efficiency for PDLS-MC and PPLS relative to the corresponding sequential versions.

The multiple independent runs strategy described in this chapter is easy to implement and requires no communication overhead during the search. This course-grained approach is ideal for execution on wide area networks with any amount of latency. The independence of the search processes means that the compute cluster can scale considerably before worrying about performance degradation due to a bottlenecks in the communication model.
4.4.2 CPPLS

In this section we explore the performance of Cooperative PPLS. Like previous sections, only sample results are shown here; please see Appendix A for full results.

We measure the scalability and speedup of CPPLS with respect to its sequential counterpart, and also against the sequential version of PLS. The plots in Figure 4.7 and 4.8 show the scalability and speedup on two of our four benchmark instances with respect to Cooperative PLS with only one search processes; e.g., one worker process which is searching and one manager process which is accepting and distributing solutions from the pool of elite solutions. Plots (a) in both figures show the speedup with respect to median run-length, while plot (b) shows the speedup with respect to median run-time. Because we did not develop a cost model for Cooperative PLS, we had to measure run-time directly and ensure all search processes were executing on identical machines. For this reason, the speedup results based on run-time is limited to 16 processors, since that is the maximum number of identical machines we could access.

The results seem to indicate that CPPLS scales better than PPLS, relative to the one process search executions. In particular the speedup in search steps on the keller6 instance is quite surprising as it appears to be super-linear. However, the reason for this result could be that (1) CPPLS does scale better than PPLS, or (2) the single-process search execution of CPPLS performs poor relative to the multi-process searches. To determine which of these two cases apply, Figure 4.9 compares the RLDs of PPLS and CPPLS on the brock800A and keller6 instances using only a single search process. These results show that the RLDs of CPPLS are relatively similar to that of PPLS (except for the keller6 instance where CPPLS outperforms PPLS), suggesting that CPPLS scales better than PPLS, at least in terms of median number of selections required to find the target clique size. Figures 4.10 and 4.11 show two sample scalability and speedup results of CPPLS with respect to the sequential version of PLS. Using this approach, we can see the scalability and speedup of CPPLS relative to PPLS. Similar to before, plot (a) is the speedup with respect to run-length, while plot (b) is the speedup with respect to run-time. The results confirm that for all but the keller6 instance, CPPLS is significantly slower. However, we were surprised to see that CPPLS seems to outperform PPLS on the keller6 instance by several orders of magnitude.

In maintaining a pool of elite solutions, one obvious design question is "how many elite solutions should be kept?". Too few elite solutions re-
duces diversity and essentially causes the algorithm to repeatedly search sub-optimal regions; too many implies there are more sub-optimal solutions to choose from, decreasing the potential of requesting a good starting point from the pool of elite solutions. Figure 4.12 shows what happens when we varied the maximum number of elite solutions on the keller6 instance and observed the corresponding performance. While there is no clear evidence indicating an optimal value, we can see that extreme ends of the parameter values displayed the worst performance.

Table 4.13 shows the results when we varied the \textit{STAGDELAY} parameter on the keller6 instance over two process configurations. As mentioned before, \textit{STAGDELAY} is calculated as \textit{STAGDELAY} = $\beta \times |K|$; thus, to see the performance impact of varying \textit{STAGDELAY} we actually varied the $\beta$ parameter. These results show that for the keller6 instance, we found the optimal value for $\beta$ to be around 100. When using other values, the performance is noticeably degraded. Setting $\beta$ too small causes the search to stop its current trajectory prematurely, which may have led to the target solution, while setting $\beta$ too high delays stagnation from being detected, resulting in unnecessary exploration of the surrounding (sub-optimal) search space.

Discussion

Our results indicate that in most cases, the performance with respect to run-length is comparable to the non-cooperative variant, although in most cases, the performance with respect to run-time is significantly worse. The most obvious explanation for this is that the extra complexity (and hence CPU time) involved in the cooperative communication mechanism outweighs the reduction in CPU time required to find the maximum clique for a given problem instance. On a positive note, very little effort was made to optimize the efficiency of the cooperative mechanisms, and thus with some code and design optimization, the costly overhead could be significantly reduced.

It should be noted that our empirical analysis is somewhat limited by the fact that we did not thoroughly explore and validate the full range of algorithmic design options for a cooperative search procedure. For example, we were unable to perform a thorough investigation on \textit{STAGDELAY} parameter, although preliminary tests have shown that the optimal value for $\beta$ on other instances was close to 100. Further empirical analysis may allow us to incorporate an adaptive mechanism wherein the value of $\beta$ is reactively updated to reflect the current state space or specific properties of the problem instance. Other future work on the cooperative search procedure
might include using a different communication scheme besides our traditional manager-worker protocol or exchanging other types of information besides the current maximum clique, such as vertex penalties.

4.5 Summary

In this chapter we showed how two state-of-the-art SLS algorithms scaled reasonably well up to 32 processors using a basic yet efficient multiple independent runs parallelization strategy. Unfortunately, the speedup and efficiency results were not quite as good as expected. One reason for this, which is a result of our original hypothesis, is that the underlying sequential algorithm's RTD does not conform to an exact exponential distribution, suggesting that it would be impossible to achieve linear speedup using the multiple independent runs parallelization strategy. However, in practice, one could argue that a target compute time could be achieved by simply adding more processors to the compute cluster, even if it meant less-than-optimal processor efficiency. It is also possible that future improvements to PLS and DLS-MC may lead to RTDs that are even closer to perfect exponential distributions, which in turn, would lead to improved parallelization speedup.

To further analyze the communication vs. computation trade-off in the multiple independent runs parallelization strategy, we introduced a basic cost model that describes the expected run-time performance while taking into account a realistic polling overhead. The polling issue is further examined in Chapter 6 where we describe a functional parallel workflow environment that uses Parallel PLS.

One improvement we could have made to the cost model estimation strategy, was to calculate $T_p$ by running the experiments on a cluster of machines, instead of the same control (reference) machine that was used to determine $T_{*r}$, since the normal environment of the polling operation is more like a large compute cluster, and not within a single machine where communication is much faster through the loop-back network interface.

In the second part of this chapter, we introduced a cooperative search strategy in which search processes share information in the form of high-quality incumbent solutions found during the search. Some believe that cooperative search strategies have become popular mainly because they allow a more natural formulation of complex algorithms, e.g., memory-based searches, where past information is saved and used throughout the search. Additional motivation for such cooperative search approaches stems from
the fact that cooperative organisms like ants, bees and even humans, have been known to accomplish certain tasks much more efficiently as a collective group than by individuals working in isolation. Although our cooperative strategy was found to be typically much slower than its non-cooperative variant, exceptional performance was observed on a particular problem instance, indicating a potential improvement in the underlying sequential algorithm. While there may be promise in further exploring cooperative variants of PPLS, for the remainder of this thesis we focus on the simple multiple independent runs version of PPLS.
Figure 4.4: Sample Scalability and Speedup results for PPLS. Plot (a) shows for varying numbers of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length.
Figure 4.5: Sample Scalability and Speedup for PDLS-MC. Plot (a) shows for varying number of processors the RLDs while (b) shows the corresponding speedup based on median run-length.
Figure 4.6: Empirical RLDs and corresponding approximated RLDs with exponential distribution for PDLS-MC on the keller6 instance.
Figure 4.7: Sample Scalability and Speedup results for CPPLS. Plot (a) shows speedup based on the median run-length, while plot (b) shows speedup based on the median run-time.
Figure 4.8: A surprising Scalability and speedup result for CPPLS. Plot (a) shows speedup based on the median run-length, while plot (b) shows speedup based on the median run-time.
Figure 4.9: Comparing RLDs for PPLS and CPPLS with 1 search process on two of our four benchmark instances.
Figure 4.10: Comparing Scalability (left) and Speedup (right) of CPPLS to PPLS. Plot (a) shows speedup with respect to the median run-length, while plot (b) shows speedup with respect to the median run-time.
Figure 4.11: Comparing Scalability (left) and Speedup (right) of CPPLS to PPLS. Plot (a) shows speedup with respect to the median run-length, while plot (b) shows speedup with respect to the median run-time.
Figure 4.12: Impact of the value of MAX_ELITE parameter on CPPLS, applied to the keller6 problem instance.
Chapter 4. Parallel Dynamic Local Search and Phased Local Search

Figure 4.13: Impact of the value of $\beta$ parameter on the RTD of CPPLS, applied to the keller6 problem instance over two search process configurations. The three $\beta$ parameter values are 50, 100 and 200; the CPPLS RTDs are compared against the PPLS RTDs with 2 and 8 search processes respectively.

63
Chapter 5

Online Phased Local Search

This chapter describes how Phased Local Search was adapted to operate in a dynamic environment where the input graph is subject to continuous modification. We present the changes made to the offline PLS algorithm which enables it to run in an online mode using trajectory continuation to resume the search with as little disruption as possible, while maintaining a high-quality incumbent solution. The algorithm is presented from both theoretical and implementation perspectives. We also provide some additional details which describe how online PLS fits within the context of a parallel workflow in which its job is to maintain a large set of maximal cliques from a dynamic graph. A detailed empirical analysis is conducted to examine its performance on a variety of hand-crafted dynamic graph series.

5.1 From Offline to Online

Our motivation behind online PLS is that of a parallel workflow which incorporates PLS, where the input graph to PLS is subject to modification at any point in time. Consider a system which is severely time-constrained, where high-quality solutions (i.e., cliques) from the updated graph must be discovered with minimal search time. In Section 3.2 we reviewed several applications which are based on solving the maximum clique problem and could be deployed in an online setting. Several of the examples used maximum cliques for feature matching and localization, a prominent problem in computer vision. Clearly, to use such an approach in an online setting, the maximum clique solver must be extremely fast if the features and landmarks are to be extracted in real-time. Another example stems from interactive applications for combinatorial optimization problems. In such interactive systems, when a user dynamically modifies parameters or specifies new constraints, the underlying search space is also perturbed. The optimization algorithm must be able to adjust accordingly and recover solutions as quickly as possible in order to provide accurate real-time feedback.
Chapter 5. Online Phased Local Search

to the user. Finally, consider a system which performs automated stock trading by constructing and analyzing a network model of the stock market using real-time data. The high-frequency nature of the data means the input graph is rapidly changing, and thus minimizing the time delay in finding maximal cliques in this dynamic graph is crucial for maximizing profits from time-sensitive opportunities. All of these examples mentioned above motivate the need for an online SLS algorithm that can adapt to dynamic changes while maintaining high-quality solutions.

While PLS was designed to find the maximum clique in a static graph, only minor modifications were necessary to make it work on a dynamic graph. When the input graph changes, two main tasks are performed. First, the data structures representing the graph are updated. At this point the search could simply restart using the new graph and proceed as if it had started an entirely new problem instance. However, this simple scheme for solving online clique finding problems does not take into account that in many cases, the changes occurring with each update of the graph are fairly small. It has been previously shown for other combinatorial problems subject to dynamic changes over time that SLS algorithms often work best in finding solutions to the corresponding series of related problem instances when not forced to restart the search process after each change [74]. Instead, it is beneficial to continue the search across changes in the input data with as little disruption as possible. This approach is known as trajectory continuation (TC) and can be easily applied to most SLS algorithms. Our online implementation of PLS exploits the power of trajectory continuation in order to consistently return a high-quality incumbent solution with minimal search cost. The pseudo-code for online PLS is given in Algorithm 4. Online PLS is realized as a wrapper around the \texttt{PLS\_search} routine, which is the original (offline) PLS algorithm with minor modifications. The two parameters \texttt{poll\_interval} and \texttt{tcs} define the termination criteria for the \texttt{PLS\_search} subroutine. Specifically, \texttt{tcs} is the target clique size, while \texttt{poll\_interval} specifies the number of search selections \texttt{PLS\_search} should perform before returning control to the main algorithm (this is the equivalent of the \texttt{max\_selections} parameter in the original algorithm; see Section 3.1 for details). After every \texttt{poll\_interval} selections, the algorithm checks for incoming “update messages”. If an update message is available, the appropriate

\textsuperscript{7}The general idea of allowing user interaction to guide an optimization algorithm as it works on a COP is called ‘Mixed-Initiative Optimization (MIO)’; see for example, Kirkpatrick et al. [90], who describe a framework for such an interactive system.

\textsuperscript{8}In our experiments the target clique size was known, but in a real-world setting it will likely not be known, and thus the termination criteria would depend only on \texttt{poll\_interval}. 

65
routines are called to maintain the dynamic graph and perform trajectory continuation. Otherwise, if no update is available, \textit{PLS.search} continues exactly as before and the search trajectory remains unchanged. The main terminating criteria for online PLS is a boolean variable called \textit{online}, which is conditioned on a total selection limit (sum of all \textit{poll.interval} selections performed in each \textit{PLS.search} invocation), or a message from an external program, such as a "shutdown" message. Recall that PLS incrementally improves on its current solution as it searches. Therefore, each time PLS pauses the search to check for an update, we use this opportunity to see if any new maximal cliques have been found since the last check, and if so, output the updated solution set. Next, the algorithm checks for dynamic update messages. Update messages come from an external source and specify a dynamic change in the input graph or parameters. There are currently three types of update messages:

- \textit{EDGE.UPDATE} — indicates updated graph edges are available;
- \textit{THRESHOLD.UPDATE} — indicates a new graph threshold has been specified;
- \textit{TERMINATE} — signals the termination of the algorithm.

A set of weighted edges $E^*_w$ is returned from the \textit{get_updated_edges} routine which is executed after a message of type \textit{EDGEUPDATE} is received. Because PLS operates on unweighted graphs, we simply use the current \textit{edge.threshold} parameter to filter out edges which do not meet the minimum weight criteria. The \textit{do_thresholding} routine is called with $E$ and $E_w$ (the edges currently in the graph and a set of updated weighted edges, respectively) to determine which edges need to be added or removed. Algorithm 5 shows this routine. If the incoming message is \textit{THRESHOLD.UPDATE}, a new edge threshold value has been specified (by an interactive user, for example), and so the original weighted edges need to be re-examined. To accommodate this dynamic change in the threshold parameter, we need to store $E_w$, the most recent edge weights for all edges in the complete weighted graph. We can then simply call \textit{do_thresholding} with $E_w$ instead of $E^*_w$, which will return the set of edges that must be added and removed to satisfy the new threshold value.
Chapter 5. Online Phased Local Search

Algorithm 4 Online PLS

1: Algorithm Online PPLS \((poll\text{-}interval, \, edge\text{-}threshold)\)
2: Input: \(poll\text{-}interval, \) initial selection limit for PLS subroutine, and
\(edge\text{-}threshold, \) the initial threshold value
3: online \(\leftarrow \) true
4: while (online) do
5: \(MC_t \leftarrow \text{PLS.search}(poll\text{-}interval, \, tcs)\)
6: // output new solutions, if any
7: if \((MC_t \setminus MC_{t-1} \neq \emptyset)\) then
8: \(\text{outputCliqueResults}(MC_t)\)
9: end if
10: \(mflag \leftarrow false\)
11: \(msg \leftarrow \text{check\_for\_messages()}\)
12: if \((msg = EDGE\_UPDATE)\) then
13: \(E_w^* \leftarrow \text{get\_updated\_edges()}\)
14: \((E_a, E_r) \leftarrow \text{do\_thresholding}(E_w^*, E, \text{edge\_threshold})\)
15: \(mflag \leftarrow true\)
16: else if \((msg = THRESHOLD\_UPDATE)\) then
17: \(\text{edge\_threshold} \leftarrow \text{get\_updated\_threshold()}\)
18: \((E_a, E_r) \leftarrow \text{do\_thresholding}(E_w, \text{E, edge\_threshold})\)
19: \(mflag \leftarrow true\)
20: else if \((msg = TERMINATE)\) then
21: \(\text{online} \leftarrow false\)
22: end if
23: if \((mflag = true)\) then
24: \(\text{add\_edges}(E_a)\)
25: \(\text{remove\_edges}(E_r)\)
26: \(\text{poll\_interval} \leftarrow \text{update\_poll\_interval()}\)
27: end if
28: end while
Algorithm 5 The do_thresholding Routine

1: **Algorithm do_thresholding**($E_w, E, \text{edge\_threshold}$)
2: **Input:** $E_w$ is a set of weighted edges, $E$ are unweighted edges currently in the graph, \text{edge\_threshold} specifies the current edge threshold value
3: **Output:** $E_a$ is the set of edges to add to the graph, $E_r$ is the set of edges to remove from the graph
4: $E_a \leftarrow \emptyset$
5: $E_r \leftarrow \emptyset$
6: for all $e$ in $E_w$ do
7: if $e.\text{weight} < \text{edge\_threshold}$ and $e \in E$ then
8: $E_r \leftarrow E_r \cup e$
9: else if $e.\text{weight} \geq \text{edge\_threshold}$ and $e \notin E$ then
10: $E_a \leftarrow E_a \cup e$
11: end for all
12: return ($E_a, E_r$)

5.2 Implementation Details

In order to work online and perform trajectory continuation, several data structures and variables need to be dynamically updated to reflect the new state of the graph. We note here that these are algorithm-specific, and that implementing trajectory continuation on a different algorithm will require maintaining a different set of data structures and variables. In order for PLS to work online without trajectory continuation, it would be sufficient to update only the following graph data structures:

- $M$ and $\overline{M}$, the adjacency and complementary adjacency matrices respectively;
- $L$ and $\overline{L}$, the adjacency and complementary adjacency lists respectively.

The actual task of trajectory continuation requires additional data structures and variables to be maintained; they are:

- $K$, the current clique in the local search;
- $C_p(K)$, the set of vertices adjacent to all but $p$ vertices in $K$. $C_p(K)$ is only maintained for $p = 0$ (the iterative improvement set) and $p = 1$ (the plateau set);
Chapter 5. Online Phased Local Search

- *missing*, a counter array that stores for each vertex the number of missing edges to the current clique \( K \). For example, \( \text{missing}[v_1]=0 \) means that vertex \( v_1 \) is adjacent to all vertices in \( K \);

- \( K_B \), the vertices of the best (largest) clique found so far during the search;

- \( K_R \), the vertices of the working clique used by the RANDOM sub-algorithm;

- \( K_D \), the vertices working clique used by the DEGREE sub-algorithm.

The actual task of trajectory continuation is realized by updating the data structures listed above, although clearly the graph data structures must be correctly updated first. For this reason, we combine the logic for updating the graph and trajectory continuation into two routines, *add_edges* and *remove_edges*.

Enhancing PLS for Sparse Graphs

The way in which we maintain the dynamic graph, vertices are never truly removed from the data structure, even when they have no edges. The vertices in the graph are ordered from 1…\( N \), where \( N = |V| \). Because we implement the graph using a \( N \times N \) adjacency matrix, it is not possible to simply ‘delete’ a vertex when it no longer has any edges, since this would require re-indexing all the vertices and subsequently re-calculating all the other data structures dependent on the vertex index. While we did not actually attempt to do this and evaluate the resulting performance, our assumption is that it would cause serious performance degradation, especially when the graph is sparse and edge additions/deletions occur frequently.

The main PLS algorithm works by incrementally updating an adjacency matrix and adjacency list, from which it can avoid choosing vertices with no edges. However, the random initialization routine chooses uniformly at random from all vertices, regardless of whether they have any edges. The reason for this is purely because the algorithm was originally designed to read DIMACS instance files which contain only vertices with at least one edge. Therefore, we made a minor modification to the initialize and reinitialize routines such that they choose a vertex uniformly at random from the set of vertices that have at least one edge. Clearly, this change introduces some additional complexity in checking each vertex degree, or maintaining a list of connected vertices, but the tradeoff becomes more favourable when the
graph is so sparse that PLS would spend a majority of the search time selecting vertices with no edges. This minor modification enables PLS to start with a connected vertex, and thus find the solution much more efficiently in a sparse graph. Figure 5.1 shows two RLDs for PLS with and without the sparse graph enhancement using one worker process. The graph under consideration was constructed from a correlation matrix of stock prices and contains 6556 vertices. Thresholding the graph edges with a threshold of 0.75 resulted in a graph with only 147 vertices which had one or more edges, a total of 764 edges and an edge density of 0.000036. The RLDs and RTDs are averaged over 200 independent runs. We can see from the plots that PLS with the sparse graph enhancement dominates the original PLS algorithm both in terms of run-length and run-time.

5.3 Adding Edges

When adding an edge \((u,v)\), it is not possible to destroy the current clique \(K\), but it may become non-maximal if \(v\) is in the plateau set and \(u\) is in the current clique, since the result is that \(v\) is then adjacent to all other vertices in \(K\). When this happens, we say that \(v\) is "promoted" to the expand list, \(C_0(K)\). Similarly, a vertex \(u\) can be promoted to the plateau list when the addition of an edge \((u,u^*)\) with \(u^* \in K\) results in \(u\) missing only one edge to \(K\). When PLS resumes after the update, if any vertices were promoted as just described, then we can guarantee that one of \(C_0(K)\) or \(C_1(K)\) are non-empty sets. Algorithm 6 shows the pseudo-code for the add_edges routine. Updating the adjacency and complementary matrix, executing the promote routine, and checking if a vertex is in the current clique are all \(O(1)\) operations, while updating the adjacency and complementary lists for each vertex has time complexity \(O(V)\). Thus the complexity of the entire add_edges routine is \(O(|E_a||V|)\).

Promoting Vertices

The promote\((v)\) routine is called when \(\text{missing}[v]\) decreases, which can happen if an edge is added between \(v\) and another vertex \(u \in K\), or if a vertex \(u_r\) is removed from \(K\) and \((u_r,v) \notin E\). Algorithm 7 shows the pseudo-code for this routine. The \(\text{missing}\) counter for \(v\) is decremented and \(v\) is moved to \(C_0(K)\) from \(C_1(K)\) (when \(\text{missing}[v] = 0\), or added to \(C_1(K)\) (when \(\text{missing}[v] = 1\)).
5.4 Removing Edges

Removing edges from the graph requires more care than adding edges, as current and best cliques can be "broken". As when adding edges, some
Chapter 5. Online Phased Local Search

Algorithm 6 The add_edges Routine
1: **Algorithm add_edges** \((E_a)\)
2: **Input:** \(E_a\), the set of edges to add to the dynamic graph
3: for all \((u,v) \in E_a\) do
   // Update graph-related data structures
   4: \(M_{uv} \leftarrow 1; \overline{M}_{uv} \leftarrow 0\)
   5: \(L_u \leftarrow L_u \cup \{u\}; \overline{L}_u \leftarrow L_u \cup \{v\}\)
   6: \(\overline{L}_u \leftarrow \overline{L}_u \setminus \{u\}; \overline{L}_v \leftarrow \overline{L}_u \setminus \{v\}\)
   // Maintain search state for trajectory continuation
   7: if \(u \in K\) then
      8: promote(v)
   9: else if \(v \in K\) then
   10: promote(u)
   11: end if
   end for all

Algorithm 7 the promote function
1: **Algorithm promote** \(v\)
2: **Input:** a vertex \(v \notin K\) to be promoted
3: \(\text{missing}[v] \leftarrow \text{missing}[v] - 1\)
4: if \(\text{missing}[v] = 0\) then
   5: \(C_1(K) \leftarrow C_1(K) \setminus \{v\}\)
   6: \(C_0(K) \leftarrow C_0(K) \cup \{v\}\)
   else if \(\text{missing}[v] = 1\) then
   7: \(C_1(K) \leftarrow C_1(K) \cup \{v\}\)
   8: \(C_0(K) \leftarrow C_0(K) \setminus \{v\}\)
   9: end if

basic internal data structures need to be updated to reflect the new graph. If an edge \((u,v)\) is removed with \(u \notin K\) and \(v \in K\), then \(u\) is demoted from \(C_0(K)\) or \(C_1(K)\). If an edge is removed connecting two vertices which are both in a working clique (e.g., \(K_B, K_R\) or \(K_D\)), we repair the broken clique by removing one of the end vertices of the removed edge. Specifically, we choose to retain the vertex with the higher degree as this will cause less disruption to \(C_1(K)\), and provide a greater chance of new vertices being promoted to \(C_0(K)\). If both vertices have the same degree, one is chosen uniformly at random. Algorithm 8 shows the pseudo-code for the remove_edges routine. Similar to the add_edges routine, updating the adjacency and complementary adjacency matrix, executing the promote and demote routines and checking to see if a vertex is in the current clique are
Algorithm 8 The remove.edges Routine

1: Algorithm remove.edges (Er)
2: Input: Er, the set of edges to be removed from the dynamic graph
3: for all (u,v) ∈ Er do
   // Update graph-related data structures
4:     Muv ← 0; Mvu ← 1
5:     Lu ← Lu \ {u}; Lu ← Lu \ {u}
6:     Lv ← Lv \ {u}; Lv ← Lv \ {v}
   // Maintain search state for trajectory continuation
7:     vm = min_degree(u,v)
8:     for all K* ∈ {KD, KR, KB} do
9:         if u ∈ K* and v ∈ K* then
10:            K* ← K* \ {vm}
11:     end for all
12:     if u ∈ K and v ∈ K then
13:        demote(vm)
14:        for all vi ∈ V do
15:           if vi /∈ K and (vi,vm) /∈ E then
16:              promote(vi)
17:          end if
18:        end for all
19:     else if u ∈ K then
20:        demote(v)
21:     else if v ∈ K then
22:        demote(u)
23:     end if
24: end for all

all O(1) operations. The complexity of repairing the working cliques (lines 8-11) depends on their respective sizes, but in the worst case they are O(|V|) operations. Updating the adjacency and complementary lists for each vertex, as well as searching for vertices not connected to vm (lines 14-18) run in time O(|V|). Thus the complexity of the entire remove.edges routine is also O(|Ea||V|).

Demoting Vertices

The demote(v) routine is similar to promote(v), but is called when missing[v] increases, which can happen if an edge (u,v) is removed such that (u,v)
Chapter 5. Online Phased Local Search

\[ G(K) \text{ or } (u,v) \notin G(K) \text{ with } u \in K \text{ and } v \notin K. \] Algorithm 9 shows the pseudo-code for this routine. The \textit{missing} counter for \( v \) is incremented, and \( v \) is moved from \( C_0(K) \) to \( C_1(K) \) (if \( \text{missing}[v] = 1 \)), or simply removed from \( C_1(K) \) (if \( \text{missing}[v] > 1 \)).

Algorithm 9 the demote function

1: \textbf{Algorithm demote} \((v)\)
2: \textbf{Input:} a vertex \( v \) to be demoted
3: \text{missing}[v] \leftarrow \text{missing}[v] + 1
4: \textbf{if} \text{missing}[v] = 1 \textbf{then}
5: \quad \textbf{if } v \in K \textbf{ then}
6: \quad \quad K \leftarrow K \setminus \{v\}
7: \quad \textbf{else}
8: \quad \quad C_0(K) \leftarrow C_0(K) \setminus \{v\}
9: \textbf{end if}
10: C_1(K) \leftarrow C_1(K) \cup \{v\}
11: \textbf{else}
12: C_1(K) \leftarrow C_1(K) \setminus \{v\}
13: \textbf{end if}

5.5 Empirical Analysis

This section contains a detailed empirical analysis of online PLS. We show how it can maintain its internal state and continue the search trajectory in order to find the maximum clique much more efficiently than a naive algorithm which restarts the search every time the graph changes.

In general, dynamic graph algorithms have been studied quite intensively in the past decade, although mainly from a theoretical perspective (see [91] for a good review of dynamic graph algorithms for undirected graphs). Furthermore, most algorithms studied in an online context tend to be deterministic algorithms, as randomized algorithms are harder to compare using competitive analysis, a theoretical measurement of the relative performance of an online algorithm against its optimal offline variant. As a result, existing literature on SLS algorithms for dynamic graph problems is scarce, and in particular no such empirical evaluations exist for the online maximum clique problem, although a recent paper provides a detailed theoretical analysis on a related (generalized) problem, the \textit{online maximum-order induced hereditary subgraph problem} [92]). Thus, due to the lack of previous research
on the online maximum clique problem, there are no established benchmark
tests that we could use to evaluate online PLS. For this reason, we generated
our own dynamic graph series for use within our empirical analysis.

Generation of Dynamic Graph Series

The first approach was to take existing DIMACS benchmark instances[93]
and to decompose them into a series of graphs. The specific benchmark
instances we used are keller4, brock200_1, and p_hat500-1, which have 400,
200 and 500 vertices respectively. These instances were chosen because
they are relatively small and thus a complete enumeration of all maximum
cliques was feasible, allowing us to measure the performance with respect to
the guaranteed optimal solution.

To start with, we created the “additive series” and “subtractive series”,
in which consecutive graphs have a monotonically increasing or decreasing
number of edges respectively. These series are formally defined as:

\[ G^{ADD} = \{G_1^{ADD}, G_2^{ADD}, \ldots, G_n^{ADD}\} \]

and

\[ G^{SUB} = \{G_1^{SUB}, G_2^{SUB}, \ldots, G_n^{SUB}\} \]

with \( n = \{5, 10, 20\} \). Furthermore, we define

\[ G_i^{ADD} = (V, E_i^{ADD}) \quad \text{and} \quad G_i^{SUB} = (V, E_i^{SUB}) \]

for \( 1 \leq i \leq n \). Thus \( V \), the set of vertices, remains constant through the
dynamic series, but the set of edges is monotonically increasing or decreasing.
The cardinalities of the edge set for each graph are ordered as follows:

\[ 0 < |E_j^{ADD}| < |E_{j+1}^{ADD}| < \frac{m^2 - m}{2} > |E_j^{SUB}| > |E_{j+1}^{SUB}| > 0 \]

for \( 1 \leq j < n \) and \( m = |V| \), which states that each graph in the additive
or subtractive series has the same set of vertices but a different set of edges
(with monotonically increasing and decreasing cardinality, respectively). The
additive series was constructed by starting with an empty graph and incre-
mentally adding edges from the original benchmark graph, and continuing
until the original graph was reconstructed. The subtractive series was cre-
ated similarly, except that instead we started with the complete graph over
\( V \) and incrementally removed edges until the original graph was revealed.
The number of edges added or removed at each stage depends on the number
Chapter 5. Online Phased Local Search

of graphs per series; e.g., for the 10-graph series, each graph contained 10% more (or fewer) edges as the previous one. Neither the empty graph or the complete graph are included in any of the series as their solution is trivial. We then merged the additive and subtractive series into a single series to produce the "mixed series", \(G^{\text{MIX}}\). More precisely, \(G^{\text{MIX}}\) is a random permutation of \(G^{\text{ADD}} \cup G^{\text{SUB}}\), resulting in the series \(\{G_1^{\text{MIX}}, G_2^{\text{MIX}}, \ldots, G_n^{\text{MIX}}\}\) for \(n = \{5, 10, 20\}\). Thus, the mixed series contains twice as many graphs as the additive or subtractive series. More importantly though, the maximum clique size and edge set cardinality in each consecutive graph is not monotonically increasing or decreasing. Figure 5.2 shows an example of how the maximum clique size changes for the brock200.1 dynamic graph series, with \(n = 40\) for the mixed series.

![Figure 5.2: Maximum clique size in the brock200.1 mixed dynamic graph series with \(n = 40\).](image)

The second approach to creating dynamic graph series is to take a correlation or similarity matrix and convert it to a complete weighted graph, and then construct a series of unweighted graphs using varying threshold levels. We use the same historical stock market data presented in [13] (and subsequently in [12]), which contains the daily closing price of 6556 US-listed securities over the time period 1998 to 2002. The average correlation coefficient for this data was shown to be approximately 0.05; for our experiment
we study the market graphs constructed by using a correlation threshold range of [0.3,0.75) with an interval of 0.05, resulting in a 10-stage dynamic graph series. The importance of creating such a dynamic graph series in this fashion is to model the application scenario described in Section 5.1 where a user dynamically changes a parameter (e.g., the graph threshold value), and in turn is presented with the corresponding unweighted graph as well as the new clique results.

In order to evaluate online PLS on our hand-crafted dynamic graph series, we needed to know the optimal clique size for each graph. The only way to guarantee an optimal clique size is to exhaustively enumerate all possible cliques. As mentioned previously, the first set of dynamic graph series used small instances so that complete enumeration was feasible. For these graphs, all maximal cliques were enumerated using the MBK algorithm [44] as an implementation was easily accessible and provided decent run-time performance. For example, the time to enumerate the GSUB series for phat500-1 and keller4 was approximately 38 minutes and 6 minutes respectively, while the brock200_1 series took over 18 hours. For comparison, our preliminary attempts to enumerate dynamic series constructed from larger but related instances (e.g., p hat1500-1, brock800_1 and keller6) did not finish executing after several days. The stock market graphs, however, contain 6556 vertices, and so clique enumeration for this series was immediately deemed impractical given our experience with the other instances described above. Instead, we resorted to using the empirical optimum, which is defined as the largest clique found by using several state-of-the-art heuristic algorithms to search for the largest clique possible, and then taking the largest clique found by any of them to be the best known solution. This approach is commonly used when a provably optimal solution cannot be determined (see [94] for another recent and related paper which uses this approach). The representative algorithms we used to determine the empirical optimum were RLS, DLS-MC and PLS. For each algorithm and each graph, we ran 100 independent trials using a very large upper limit on search selections (30 million) or time (1 hour).

5.6 Results

In the following, we compare PLS without trajectory continuation to PLS with trajectory continuation on the set of dynamic graph series previously described. The statistical significance of our results are also verified using the Mann-Whitney U test.
5.6.1 Synthetic Dynamic Graph Series

For each stage $G_i$, we show that the search cost is lower when continuing the search instead of restarting from scratch. Although we provide results for the additive, subtractive and mixed series, we focus more on the mixed series. The rational is that in practice one would expect the dynamic graph to have characteristics similar to the mixed series rather than the additive or subtractive series, in the sense that edges can be added or removed between consecutive graphs, rather than strictly one or the other. Full results for the additive and subtractive series are given in Appendix B.

Figure 5.3 summarizes the speedup results for all the mixed graph series, showing that trajectory continuation does indeed decrease the time to reach an empirical optimal solution after the underlying graph has changed. For brevity, Tables 5.1, 5.2 and 5.3 summarize the performance over the additive, subtractive and mixed series for each of the three instances. Each table shows the total (cumulative) search selections needed to find the maximum clique at every stage through the series. Like usual, the total search selections is an average over 100 independent runs.

Table 5.4 shows the number of stages for each series for which the optimal clique size changes. The results show that the optimal clique size changes infrequently in the additive series, most of the time in the mixed series, and always in the subtractive series. The plots in Figures 5.4, 5.5 and 5.6 show how the online algorithm performs on three dynamic series of the same
Chapter 5. Online Phased Local Search

### Table 5.1: Search step improvement using trajectory continuation on additive series.

<table>
<thead>
<tr>
<th>Instance</th>
<th>G*ADD</th>
<th>5-stage Series</th>
<th>10-stage Series</th>
<th>20-stage Series</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No TC</td>
<td>TC</td>
<td>No TC</td>
<td>TC</td>
</tr>
<tr>
<td>brock200_1</td>
<td>2093</td>
<td>1844 1.14</td>
<td>29243 1.08</td>
<td>63525 2.15</td>
</tr>
<tr>
<td>keller4</td>
<td>1190</td>
<td>1284 0.93</td>
<td>901 1.60</td>
<td>1846 2.29</td>
</tr>
<tr>
<td>p_hat500-1</td>
<td>745</td>
<td>444 1.68</td>
<td>7347 1.66</td>
<td>14263 2.81</td>
</tr>
</tbody>
</table>

### Table 5.2: Search step improvement using trajectory continuation on subtractive series.

<table>
<thead>
<tr>
<th>Instance</th>
<th>G*SUB</th>
<th>5-stage Series</th>
<th>10-stage Series</th>
<th>20-stage Series</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No TC</td>
<td>TC</td>
<td>No TC</td>
<td>TC</td>
</tr>
<tr>
<td>brock200_1</td>
<td>4955</td>
<td>2667 1.86</td>
<td>21654 1.71</td>
<td>56246 1.25</td>
</tr>
<tr>
<td>keller4</td>
<td>1213</td>
<td>489 2.48</td>
<td>7850 1.30</td>
<td>8505 1.44</td>
</tr>
<tr>
<td>p_hat500-1</td>
<td>10027</td>
<td>8880 1.16</td>
<td>30683 1.44</td>
<td>68345 1.39</td>
</tr>
</tbody>
</table>

### Table 5.3: Search step improvement using trajectory continuation on mixed series.

<table>
<thead>
<tr>
<th>Instance</th>
<th>G*MIX</th>
<th>10-stage Series</th>
<th>20-stage Series</th>
<th>40-stage Series</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No TC</td>
<td>TC</td>
<td>No TC</td>
<td>TC</td>
</tr>
<tr>
<td>brock200_1</td>
<td>7666</td>
<td>4227 1.81</td>
<td>47682 1.2</td>
<td>123777 1.20</td>
</tr>
<tr>
<td>keller4</td>
<td>2745</td>
<td>2003 1.37</td>
<td>3068 1.16</td>
<td>11357 1.64</td>
</tr>
<tr>
<td>p_hat500-1</td>
<td>14338</td>
<td>7330 1.96</td>
<td>42310 1.66</td>
<td>84189 1.61</td>
</tr>
</tbody>
</table>

### Table 5.4: Number of stage transitions in which the maximum clique size changes.

<table>
<thead>
<tr>
<th>Instance</th>
<th>5</th>
<th>10</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add</td>
<td>Sub</td>
<td>Mixed</td>
</tr>
<tr>
<td>brock200_1</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>keller4</td>
<td>3</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>p_hat500-1</td>
<td>3</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

Table instance but with a different number of stages. The corresponding plots for the other instances can be found in Appendix B. As can be seen in the plots, PLS with TC (the lighter line) requires few search selections than PLS without TC (darker line) to find the target clique size at most stages through the series. The most obvious advantage for PLS with TC occurs...
when the target clique size does not change between successive stages. All
the results presented above show the performance over the entire series. For
thoroughness, we also show some representative RLDs for individual graphs
from each series. The plots in Figure 5.7 show some typical RLDs for the
additive series with \( n = 20 \). In particular, plot (a) is illustrative of cases
when PLS with TC requires very few (or none) search selections to find the
target clique size. On the other hand, plot (b) is illustrative of RLDs for
stages in which there is no clear advantage for either PLS with or without
TC. The plots in Figure 5.8 show two typical RLDs for the brock200_1 and
keller4 subtractive series with \( n = 20 \). In general, it can be seen that PLS
with TC outperforms PLS without TC.

5.6.2 Mann-Whitney U Test

To further validate the statistical significance of our tests, we use the Mann-
Whitney U test to compare the RLDs for PLS with and without TC for each
graph in a given series. In particular, we test the null hypothesis that the
distributions are not equal and differ with respect to their medians.

The Mann-Whitney test is calculated as follows. First, calculate \( U \):

\[
U = n_1 n_2 + \frac{n_1(n_1 + 1)}{2} - R_1
\]

where \( n_1 \) and \( n_2 \) are the two sample sizes, e.g., the number of independent
runs performed on each graph. \( R_1 \) is the sum of ranks in sample 1, which is
calculated by arranging both sets of samples (run-lengths) in a single ranked
series, and then adding up the ranks corresponding to the items from sample
1.

In our experiments we perform 100 independent runs for each experi-
ment. When performing the U test, if sample sizes are greater than 20, \( U \)
is approximately normal and the standard normal distribution is used with

\[
z = \frac{(U - \frac{n_1 n_2}{2})}{\sqrt{\frac{n_1 n_2(n_1 + n_2 + 1)}{12}}}
\]

\( z \)'s significance can be checked in a normal distribution table to validate
the null hypothesis. For reporting results, we use \( \alpha = 0.5 \) as the significance
level at which to accept or reject the null hypothesis.

Table 5.5 shows the percentage of stages in each series for which the
difference in the median run-length of PLS with and without TC were significantly different; in other words, the percentage of U tests for which the null hypothesis was rejected.

<table>
<thead>
<tr>
<th>Dynamic Graph Series</th>
<th># of stages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 20 40</td>
</tr>
<tr>
<td>brock200.1</td>
<td>20 50 65</td>
</tr>
<tr>
<td>p_hat500-1</td>
<td>20 70 63</td>
</tr>
<tr>
<td>keller4</td>
<td>30 55 70</td>
</tr>
</tbody>
</table>

Table 5.5: Percentage of stages for which the null hypothesis was rejected with $\alpha = 0.5$.

5.6.3 Stock Market Dynamic Graph Series

We now evaluate our online algorithm using a dynamic graph series constructed from real data as described in Section 5.5. Table 5.6 shows the online performance for the resulting dynamic graph series. The corresponding performance graphs showing individual and cumulative selections at each stage can be found in Appendix B.

<table>
<thead>
<tr>
<th>Series</th>
<th># Stages</th>
<th># Search steps</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>No TC</td>
<td>TC</td>
</tr>
<tr>
<td>add</td>
<td>10</td>
<td>8355</td>
<td>3009</td>
</tr>
<tr>
<td>remove</td>
<td>10</td>
<td>8336</td>
<td>1920</td>
</tr>
<tr>
<td>mixed</td>
<td>20</td>
<td>16683</td>
<td>5769</td>
</tr>
</tbody>
</table>

Table 5.6: Performance differences between PLS with and without TC on the dynamic market graph series.

Figure 5.9 shows how the maximum clique size changes throughout the dynamic graph series. The plot shows the trace through the additive, subtractive and mixed series. The plots in Figure 5.10 show the number of edges (a) and edge density (b) with respect to the correlation threshold from the stock market dataset used to construct the dynamic market graph series. Previous research on the market graph tends to exhibit small-world characteristics such as the power-law (scale-free) property [18]. Figure 5.11 shows the vertex degree distributions for the graphs corresponding to correlation thresholds $t = 0.3$ and $t = 0.6$. For brevity we omitted the other graphs for thresholds within the interval $(0.3, 0.6)$, but we note that their distributions were nearly identical to those shown here. The fact that the distribution
plots are relatively stable for various threshold values and can be approximated be a straight line confirms that our dynamic market graph series is also characterized by a power-law distribution.

5.6.4 Discussion

One assumption that was made when considering this approach is that the changes to the underlying search space can be expected to be relatively minor, and thus continuing the search is more efficient than starting from scratch. In other words, we expected online PLS to perform better when the distance between the two consecutive graphs is minimal, where distance refers to the number of edge additions or removals required to transform graph $G_i^{DYN}$ to $G_{i+1}^{DYN}$. As it turns out, our results show that recovering a new solution from an old one is almost always beneficial, regardless of the severity of the graph change. Although the speedup afforded by the online adaptation is not overwhelming, when operating in a highly time-constrained environment, every bit of speedup counts. Furthermore, even fairly drastic changes to the graph often have little or no affect on the maximum clique. Table 5.4 shows how the maximum clique size changes throughout each of the dynamic graph series; we can see that in several series, there are numerous graph transitions in which the maximum clique size does not change. In order to validate this assumption, we created dynamic graph series with varying number of stages. Recall the when creating the dynamic series, we added a constant percentage of edges at each stage; in other words, more stages means fewer edge additions or removals at each stage. The results shown in Table 5.5 validate our assumptions, showing that the number of improvements which were statistically significant increases with the number of stages in the dynamic graph series. The actual speedup, however, seems to have no correlation with the number of stages. Figure 5.3 shows the speedup when using TC for the three sets of graph series, and a visual inspection shows there is no clear correlation between the observed speedup and the number of stages.

The results from the market graph series show that online PLS with TC tends to perform better than online PLS without TC on graphs constructed from real-world data. One possible explanation for this points to the fact that vertex degrees through our market graph series exhibit a power-law distribution (see Figure 5.11). In such a graph, we can expect the maximum clique to be found in one of the dense areas. Accordingly, previous research on market graphs has shown they exhibit such scale-free properties [10]. Therefore, assuming a relatively minor modification to the graph, we might
Chapter 5. Online Phased Local Search

expect that the new solution (i.e., the new maximum clique) would be in the same dense region of the search space, thus allowing trajectory continuation to more easily converge to the new solution.

5.7 Summary

In a dynamic environment, the ability to operate online is a clear advantage for any algorithm which must return near-optimal solutions within a fixed time frame. In particular, we consider applications which run in a real-time environment, and emphasize that time is often highly constrained and near optimal solutions are always preferred to no solutions at all (or more likely, the possibility of receiving low-quality solutions). For this reason, online stochastic local search algorithms offer substantial advantages over traditional complete search methods as they are able to quickly recover high-quality solutions in a dynamically changing problem space.

In this chapter we presented an online implementation of Phased Local Search which uses trajectory continuation to resume the search instead of restarting from scratch each time the input graph changes. Our results confirm that the trajectory continuation method reduces the number of search steps (and hence the run-time) needed to find the target clique size in a series of related graphs, and that its performance improvement over the non-trajectory continuation variant is most significant when the perturbation to the input graph is relatively minor.

By combining the scalable parallelization strategy as discussed in Chapter 4 with the online searching capabilities described in this chapter, our algorithm has the power to operate in highly dynamic and time-sensitive environments such as the stock market.
Figure 5.4: Online performance results for the p_hat500-1 mixed series with \( n = 10 \); the top figure is selections per graph, the bottom figure is cumulative selections.
Figure 5.5: Online performance for the p.hat500-1 mixed series with $n = 20$; the top figure is selections per graph, the bottom figure is cumulative selections.
Figure 5.6: Online performance for the p.hat500-1 mixed series with $n = 40$; the top figure is selections per graph, the bottom figure is cumulative selections.
Figure 5.7: Two different but typical RLDs for the additive series with $n = 20$. 
Figure 5.8: Typical RLDs for the subtractive series with $n = 20$. 
Figure 5.9: Maximum clique size trace through the dynamic market graph series.
Figure 5.10: Edge count and edge density as a function of correlation threshold.
Figure 5.11: Vertex degree distributions on the dynamic market graph series for correlation thresholds $t = \{0.3, 0.4, 0.5, 0.6\}$. 
Chapter 6

The Parallel Workflow

In this chapter we investigate the design and implementation of a fully distributed parallel workflow environment for problems in which high-volume, high-frequency time series must be analyzed online and in (near) real-time. The system we propose can perform online correlation analysis and clustering of thousands of variables within high-frequency time series, offering opportunities for automated control systems and knowledge discovery within a broad range of application domains such as finance, bioinformatics and physics. The correlation analysis is performed using Maronna [16], a robust correlation method amenable to parallelization and online computation, while clustering is achieved using Online Parallel PLS (OPPLS), our clique-finding algorithm described in Chapters 4 and 5.

Typical of many real-world problems, data processing and subsequent analysis is becoming increasingly complex as the size and frequency of data increases. Furthermore, much of this data is now being offered through real-time streams, presenting yet another level of complexity. In many cases, the frequency and distribution of incoming data is not known a priori, and even the smallest change in the input data has the potential to dramatically perturb the underlying problem space. Clearly, re-computing the solution from scratch each time there is a small change is not only a waste of valuable computing resources, but also detrimental to performance in a system where time is already severely constrained. Our system addresses this issue by using novel online algorithms that are designed to operate efficiently on dynamic data streams, resulting in more accurate and timely information dissemination.

Given the sporadic nature of the data as just described, the challenge was to design a system with the following properties:

- The system must be able to operate on a real-time flow of data and thus the algorithms in the system must be online and attempt to minimize the calculation for each of the steps.

- The system must be able to scale to the size of the input data for the computationally intense methods such as correlation and clique-
finding, and obtain high quality solutions in a reasonable amount of time.

- The system must be robust to noisy, missing and unpredictable data.
- The system must be flexible in order to support a variety of user-defined configurations.

For example, consider stock trading systems that must process, in real-time, the trade and quote information for all the stocks on a given exchange. Furthermore, consider new electronic exchange integration hubs such as INET that offers direct access to over 7,200 US exchange-listed securities, with approximately 65 million quotes and 5.7 million trades per day [95].

The goal of the system is to correlate and cluster high-frequency data streams such as the ones just described in order to construct and analyze dynamic graphs in real-time. The output from this real-time analysis can be redirected to various components, such as a visualization client, decision support system, or a closed-loop automated trading system.

6.1 System Design and Architecture

At the basic level, our system utilizes a generic pipeline processing architecture called the Pipeline Processor Farm (PPF) [96], which aims to increase throughput and reduce latency by exploiting the fact that the system can be decomposed into independent components. The PPF architecture was first introduced to parallelize real-time embedded image processing applications, and has since been used in a variety of real-time applications such as completion detection in dynamic CMOS logic [97] and real-time OCR applications [98].

Figure 6.1 shows a generalized system architecture diagram of the distributed communication model we employ within our framework. The entire system is first split into two parts: the control process and controlled processes. The controlling process is a single (sequential) process which can be controlled externally using standard TCP/IP sockets. This allows a direct point of communication between an external network and the distributed components. The remaining system processes are further divided into their respective groups, which are also referred to as the compute stages. Within each compute stage, a single process is designated as the leader, and is responsible for communicating with the control process and relaying messages to the other processes in the stage.
Chapter 6. The Parallel Workflow

The system is composed of five main programs (components), two of which perform the bulk of the computation. We briefly summarize each of the components below.

- **Data Collector**: Reads data from a variety of external sources, including files, databases and live feeds.

- **Filter and Accumulator**: Receives individual data items from the Data Collector, performs any necessary data cleaning, and prepares data samples to be sent to the correlation component.

- **Parallel Robust Correlation**: This is the first major component, and conforms to a processor farm model. The manager process in this component receives the input data samples and decides how many new correlation values must be calculated. It then continuously assigns worker processes to tasks until the correlation calculation is complete, where a ‘task’ is defined as computing a single correlation value. Refer to Section 6.3 for details.

- **Parallel Clique-based Clustering**: This is the second major component. The manager process receives updated correlation coefficients, which are used to update the pair-wise similarity measures for the data points being clustered. Refer to Section 4 for details.

- **Results Analysis Server**: This is the final component in the pipeline and comprises only a single thread of execution. This component is

![Figure 6.1: Topology of process structure and communication model](image)
6.1.1 Inter-process Communication

Inter-process Communication (IPC) is an important design criteria for any distributed system. There are several factors that affect the decision, including target architectures, run-time environments, latency requirements and shared memory requirements. For communication within and between system components, we have chosen to use LAM and MPICH, two open source implementations of the Message Passing Interface (MPI), the de-facto standard API for message passing in parallel computing. We note that although our implementation uses these open-source packages, it should run on any MPI implementation. Due to its openness and wide-spread usage, MPI runs over most network infrastructures including TCP/IP, allowing the system components to be effortlessly distributed over a wide-area network (WANs). Furthermore, the LAM and MPICH implementations have recently been redesigned to work with Stream Control Transmission Protocol (SCTP) [99] to reduce the effect of latency on task farm programs such as the ones used in our workflow.

6.1.2 Pipeline Architecture

Our system is embodied within a pipeline architecture, in which data flows downstream, and at each stage in the pipeline, output from one component becomes input to the next. The pipeline is comprised of independent programs working in parallel on different sets of data, thus conforming to a Multiple Program Multiple Data (MPMD) model. The datasets, however, are related, in a sense that they all originated from the same source, except that the data is transformed (e.g., a correlation matrix is created from the data samples) or abstracted (e.g., correlation coefficients map to edges in the dynamic graph model) as it passes through the various stages.

6.1.3 Processor Farm Architecture

The two main parallel components within our system utilize a processor farm architecture. Figure 6.2 (a) shows a generic processor farm where a manager "farms" out tasks to the worker processes until the entire computation is complete. In the case of parallel Maronna, worker processes are given a batch of jobs to compute, where a job corresponds to calculating a single
correlation co-efficient. Upon completion the worker sends back the batch of results and subsequently asks for another batch of jobs. This continues until the correlation computation is complete. In OPPLS, each worker process searches the dynamic market graph for maximal cliques. After performing a given number of search steps, the worker checks for a message from the manager which is either a graph update or termination notice. If no message is present the worker performs another round of searching. The number of search steps performed by each worker process depends on the load levels of the executing machine's CPU. An adaptive load-balancing scheme for estimating the number of search steps is described in Section 6.4.4. Both components employ a course-grained communication model which makes it convenient to overlap communication with computation, thus hiding the effects of latency (See Figure 6.2 (b)).

6.1.4 Mapping Processes to Processors

The design of the communication protocol used between OPPLS and Maronna allows us to overlap the two stages onto the same processors. This is possible because of their inter-dependence on the data. In other words, when one task is executing, the other need not be, thus guaranteeing that only one of the two tasks is utilizing the system resources at any given time. For example, when Maronna is computing correlation values, OPPLS is waiting to receive those updated values, and when the correlation task is complete, OPPLS searches for cliques in the new graph while Maronna waits for the next batch of data samples. As previously mentioned, OPPLS performs a given number of iterations before checking to see if Maronna is ready to start working again. If a message from Maronna is present, OPPLS terminates
the search and releases the compute resources; otherwise, OPPLS continues searching for improved clique solutions.

6.2 Time Series Data

In this thesis we consider the time series consisting of a set of $M$ data points $P = \{p_0, p_1, \ldots, p_{M-1}\}$ sampled from a function $F$ over some time period. Each data point $p_i$ is a tuple of the form $(d_i, \tau_i)$, where $\tau_i$ is the time stamp and $d_i = F(\tau_i)$, the value of the function sampled at time $\tau_i$. We also assume here that the values of $\tau_i$ span a finite time window of $T$ time units.

In the case where a data stream contains time series data for $N$ variables, we can partition the data points into $N$ separate time series $D_i$, for $i = 1, \ldots, N$, giving one $D_i$ for each variable $x_i \in X$, where $X = \{x_1, \ldots, x_N\}$ denotes the set of all variables. $D_i$ is thus defined as:

$$D_i = \{(d_{ik}, \tau_{ik}) \mid 0 \leq k \leq K_i\} \quad (6.1)$$

with $d_{ik}$ and $\tau_{ik}$ representing the $k$-th data point and time stamp, respectively, for $x_i$. $K_i$ is the total number of data points for variable $x_i$.

Because the system is operating in an online mode, data samples are continuously arriving, and the total number of samples for each variable used in the correlation analysis is also increasing. If all $K$ samples for each variable were kept indefinitely, then the correlation patterns under consideration would span the entire time period associated with the data samples. In our system, we are interested in recent, short-term patterns, and so our correlation analysis uses only the most recent data samples. Specifically, our solution is to keep a finite-sized first-in-first-out (FIFO) queue of data samples for each variable (see Figure 6.3).

The set of data samples in the queue (of length $Q$) for variable $x_i$ at time $t$ is $D_{i(t)} \subseteq D_i$ and is defined as

$$D_{i(t)} = \{d_{ik} \mid \tau_{ik} \in [t - Q\Delta t, t], 0 \leq k \leq Q\} \quad (6.2)$$

The data samples $D_{i(t)}$ fall within the time window $[t - Q\Delta t, t]$, which is the last $Q$ samples taken with sampling interval $\Delta t$. All data points are ordered by their time stamp (where resolution of time depends on context) and there may be several data points with the exact time stamp, thus holding only a weak inequality condition $\tau_k \leq \tau_{k+1}$. Aggregation and interpolation methods can be used to deal with multiple or missing data points within a time window; an example of this applied to stock market data is presented.
New data samples enter the queue, old ones are removed

Figure 6.3: Data sample queues for all $N$ variables at time $t$.

in Section 7.2. When using a synchronous sampling strategy, sampling of the data stream is performed at a regular interval of $\Delta t$ time units. The matter is complicated when asynchronous sampling is performed. The next section discusses this issue.

Asynchronous Sampling

At a given time $t$, the set $X^{(t)} \subseteq X$ denotes all variables for which at least one data sample arrived within the last time window $[t - \Delta t, t]$. More formally, we define $X^{(t)}$ as

$$X^{(t)} = \{ x_i \in X | \exists (a_{ij}, r_{ij}) \text{ s.t. } r_{ij} \in [t - \Delta t, t] \}$$

(6.3)

When using a synchronous sampling strategy, $X^{(t)} = X$, since there will be a new data sample for each variable (either a new data item was received, or one was created via interpolation). However, when using asynchronous sampling, if a variable $x_i$ does not receive a data sample during the current time window (i.e. $x_i \notin X^{(t)}$), then its queue will remain unchanged. If
the sample queues for two variables \( x_i \) and \( x_j \) do not change between successive time windows, then their correlation coefficient \( \rho_{ij}^{(t)} \) will not change either. The exact number of correlation values which need updating can be calculated using Equation 6.4.

By only using the subset of variables \( X^{(t)} \) in each new correlation calculation, our system has the ability to correlate an event-series, as opposed to a time series, where an "event" is the data sample, and the exact meaning depends on the context. This approach allows us to correlate events that are occurring opposed to the lack of events within some time interval.

6.3 Correlation Calculation

Calculating the correlation between a collection of random variables is a standard technique used to determine the strength of a linear relationship between variables. A correlation matrix records the correlation coefficients between all pairs of variables. Unfortunately, correlation analysis is very sensitive to the presence of outliers in the data. Our solution to this problem is to use a robust correlation technique called the Maronna method [16]. The computational time complexity of the Maronna method is \( \mathcal{O}(MN^2) \) for \( N \) variables and \( M \) data samples. However, as shown by Chilson et al., it can be easily parallelized and scales to large numbers of processors [100]. The Maronna method can be structured into an embarrassing parallel algorithm where each pairwise correlation coefficient can be calculated independently. The pairwise correlation calculation is an iterative process with two control parameters that allow us to specify the accuracy of the correlation as well as the maximum number of iterations. These two controls can be used reduce the computation time, offering a flexible trade-off between the precision of the results and response time. Details on the Maronna method and its parallelization are described in [16] and [100], respectively.

6.3.1 Maintaining the Real-time Correlation Matrix

As data samples enter the system we can re-compute the correlation coefficient between variables \( x_i \) and \( x_j \) at time \( t \) by calling the following function:

\[
\rho_{ij}^{(t)} = \text{maronna.correlation}(D_i^{(t)}, D_j^{(t)}, n, \epsilon, \text{limit})
\]

where \( D_i^{(t)} \subseteq D_i \) and \( D_j^{(t)} \subseteq D_j \) are the data samples in the queue for variables \( x_i \) and \( x_j \) at time \( t \), and \( n = \min(|D_i^{(t)}|, |D_j^{(t)}|) \). The parameter
Chapter 6. The Parallel Workflow

c is the precision at which the iteration of the correlation of two variables stop, while limit specifies the maximum number of iterations. As previously mentioned, these control parameters can be used to fine-tune the level of precision. In order to avoid insignificant correlations, we only calculate correlations coefficients between two variables $x_i$ and $x_j$ if they have a minimum number of data samples in their queues (in our experiments we use 50 as the minimum).

When a new data sample is received for variable $x_i$, its correlation coefficients with all other variables need to be updated; that is, we need to re-compute $\rho_{ij}^{(t)}$ for all $j \in \{1, ..., N\} \setminus \{i\}$. Recall that $X^{(t)}$ is the set of variables that received at least one data sample within the last time window. Now, let $M^{(t)} = |X^{(t)}|$, then we can calculate $\zeta^{(t)}$, the exact number of correlation coefficients that will need updating at time $t$ as follows:

$$\zeta^{(t)} = \frac{N(N - 1)}{2} - \frac{(N - M^{(t)})(N - M^{(t)} - 1)}{2} = M^{(t)}N - \frac{M^{(t)}(M^{(t)} - 1)}{2}$$  \hspace{1cm} (6.4)

An interesting observation is that as $M^{(t)}$ increases, there is less duplicated computation, since the correlation coefficients between the variables within a batch are calculated only once. Furthermore, as $M^{(t)}$ decreases, the resolution at which the correlation matrix is updated becomes finer. For example, if $M^{(t)} = 1$, then the correlation matrix is updated after each new data sample. Unfortunately, given the high-frequency nature of the data streams under consideration, it is not efficient to perform this calculation on a per-update basis. Therefore, we define a time window where data is ‘batched’ and subsequently sent to the Maronna stage. The Maronna stage then broadcasts the updated samples to the workers, and the processor farm is started. A manager process dispatches correlation tasks to workers who perform the independent correlation computation and return the result. Thus, there is a trade-off between having a small batch size that results in some wasted calculation but faster response time, versus larger batch sizes over larger time windows that give a slower response time. Depending on the application context, the response time in updating the correlations might need to be minimized; e.g., a non-interactive automated stock trading agent that relies on sub-millisecond response times, versus an interactive visualization system which may only require correlation updates every few seconds.
Chapter 6. The Parallel Workflow

6.3.2 Communication of Correlation Updates

Recall that each variable \( x_i \in X \) corresponds to a vertex \( v_i \in V \) in the dynamic graph, and \( \rho_{ij}^{(t)} \), the correlation value between variables \( x_i \) and \( x_j \) at time \( t \), is used to determine if the edge \((i,j)\) is present in the dynamic graph. Thus, each time a correlation coefficient is recalculated the corresponding edge in the dynamic graph needs to be re-examined.

Equation 6.3 defines \( X^{(t)} \) as the number of variables with at least one data sample in the time window \([t - \Delta t, t]\), while Equation 6.4 defines \( \zeta^{(t)} \) as the total number of correlation coefficients that need to be re-calculated. Mapping these to into graph notation, we obtain \( V^{*} \subseteq V \), the set of vertices for which at least one edge was affected, and \( E^{*} \subseteq E \) with \( |E^{*}| = \zeta^{(t)} \), the set of edges corresponding to the updated correlation coefficients. In fact, the edge set \( E^{*} \) corresponds to the edges of \( G(V^{*}) \), the complete subgraph induced by \( V^{*} \). Therefore, after each correlation update the complete subgraph \( G(V^{*}) \) needs to be re-examined by OPPLS in order to update its dynamic graph model (see Section 5.2 for details).

6.4 Clique-based Clustering

Clustering of the dynamic graph is achieved by finding a large set of maximal cliques using the OPPLS algorithm described in Chapters 4 and 5.

6.4.1 Thresholding

In order to limit the size of the graph, and to focus on highly correlated variables, PLS removes edges corresponding to correlation coefficients that fall below a user-specified threshold. In our experiments, the thresholding process uses the absolute value of the correlation coefficients; that is, given a threshold value \( \text{thresh} \in [-1, 1] \), the thresholded graph has an edge set \( E_{\text{thresh}} \), where \( e \in E_{\text{thresh}} \iff |\text{weight}(e)| \geq t \). Of course, there may be situations where one is only interested in correlations strictly greater or less than a given threshold; e.g., \( e \in E_{\text{thresh}} \iff \text{weight}(e) \geq t \) or \( e \in E_{\text{thresh}} \iff \text{weight}(e) \leq t \).

6.4.2 Online Parallel PLS

In a realistic setting, where the graph changes dynamically in response to real-time data input or user interaction, maximum clique sizes will not be known a priori. Therefore, rather than specifying a target clique size, we
Chapter 6. The Parallel Workflow

run each PLS process for the same amount of time and subsequently determine which of them have achieved the largest clique size. As a consequence of the high-frequency data monitored by our system, the graph analyzed by PLS is subject to frequent modifications. In response to any such modification, we need to solve the clique finding problem for the new graph before further changes may invalidate the new solutions. In principle, the scalable parallelization of PLS allows us to address this challenge in a very simple manner, namely by solving the problem instance arising from each modification of the graph by performing parallel PLS on sufficiently many processors. Furthermore, SLS algorithms such as PLS have a desirable 'anytime property', in that at any point during the search they can return the best candidate solution found so far. Therefore, even if parallel PLS has not found a maximal clique before the graph changes, it can produce a meaningful suboptimal solution. However, this simple scheme for solving online clique finding problems does not take into account that in many cases, the changes occurring with each update of the graph are fairly small. Instead, we use trajectory continuation to continue the search across changes in the input data with as little disruption as possible. Parallel and Online PLS are described in detail in Chapters 4 and 5, respectively.

6.4.3 Recording Sets of Maximal Cliques

Our final modification to the PLS algorithm is motivated by the fact that in our clique-based analysis of the dynamic graph, we are interested in finding not just a single maximum clique, but a large and diverse set of maximal cliques. As a first way of addressing this goal, we modified online parallel PLS to record the set of all maximal cliques encountered since the initialization of the search or last change in the dynamic graph. While it would be relatively easy to extend the mechanism used for checking and repairing the current and best cliques after each change in the graph, further empirical analysis is required to determine whether the computational cost of this process is amortized.

Currently, the set of maximal cliques collected by OPPLS are combined into the so-called 1-clique graph (an instance of a k-clique graph), whose vertices corresponds to those involved in at least one maximal clique and whose edges are precisely those found in the set of maximal cliques [101]. We have a simple visualization client that can be used to explore this graph. Clearly, the clique data collected by our algorithm contains additional useful information that can be extracted by further analysis. For example, we have now started analyzing the clique overlap structure, which has been shown
to exhibit small-world scaling properties [102].

### 6.4.4 Polling for Messages

Because information is constantly flowing through the pipeline, OPPLS only needs to search for cliques while the information it has is ‘up-to-date’. Figure 6.4 shows the communication timeline between Maronna and OPPLS. In order to implement this scheme, OPPLS needs to poll for messages from Maronna that indicate a correlation computation is commencing. Similar to the polling issue described in Section 4.1.2, there is a trade-off between polling too often and slowing down the search, and polling too little resulting in unnecessary searching. For example, if the incoming data frequency is high enough such that Maronna is continuously working, OPPLS performs only a minimal number of search steps between successive graph updates. However, if the incoming data frequency is low and Maronna is idling (waiting for data), then OPPLS continues to refine the clique results until new data is received.

Algorithm 8 shows our proposed solution, which is an adaptive polling mechanism that dynamically adjusts the polling interval based on the recent incoming data frequency. The parameter $m$ is the number of consecutive times OPPLS stopped searching to check for an update message. This parameter informs the adaptive polling mechanism that incoming data frequency is decreasing, and thus OPPLS should increase the polling interval. If $m = 0$, then a message was waiting as soon as OPPLS finished its first
poll\_interval selections, indicating that the data frequency is increasing and that poll\_interval should be decreased. The exact function we use to adapt poll\_interval has not been optimized or fully evaluated, but initial experiments suggest that it maintains a reasonable value for the polling interval with respect to the incoming data frequency. Figure 6.5 shows a trace of the poll interval parameter value from an experiment using a 5 second sampling interval. It can be seen that there is some fluctuation in the parameter value as it tries to adapt to the varying data input frequencies, however the range of values is relatively stable considering how many search selections the OPPLS algorithm can perform per second.

6.5 Evaluation

This section provides system performance and scalability results using synthetic data. The results show how our intelligent parallel architecture enables state-of-the-art parallel methods to scale-up by simply by increasing the number of processors.

For synthetic data testing, we created a simple multivariate time series data stream generator. The data stream generator can be made to produce arbitrary-sized subsets of variables that are highly correlated with the remaining variable exhibiting random values. The chosen subsets of variables form a maximal clique in the dynamic graph, which OPPLS detects and outputs to its downstream component (the Results Analysis Server).

We tested the scalability and throughput of the system by varying the number of processors and measuring the time delay of the data passing
through the pipeline. Our current testing environment is a small compute cluster comprised of 14 dual Intel Xeon 3.06GHz CPU's with 2GB RAM, and another small cluster with 6 dual Intel Xeon 2.0GHz CPU's with 4GB RAM. We tested the scalability and throughput of the system by varying the number of processors and measuring the time delay through the pipeline. Figure 6.6 shows the result of executing the system with 5, 10 and 18 processors on Maronna and PLS. We do not include the processors executing the single processes corresponding to the first two stages, the last stage and the control process. The batch size specifies how many variables receive new data values at each sample period. By setting the value of batch size equal to the number of variables, we are simulating a complete synchronous sampling; i.e., all $N$ variables received new data during each sample period. The response time increases at the beginning of each run because as the input queue for each stock fills, the size of the samples for correlation increases. Eventually the queues fill, after which the sample sizes remain constant.

Table 6.1 shows the speedup results of the average response times for each of the series shown in Figure 6.6. Table 6.2 reports several response time statistics for various processor configurations. It is worth noting that the standard deviation (stddev) and variance (var) of the response times decrease when more CPU's are used — a result of the inherent load-balancing
Figure 6.6: Response time for a dataset with 2000 variables, batch size 2000, and three processor configurations.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Avg. response time (seconds)</th>
<th>Speedup (w.r.t. 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>14.49</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>6.83</td>
<td>2.12</td>
</tr>
<tr>
<td>18</td>
<td>3.74</td>
<td>3.87</td>
</tr>
</tbody>
</table>

Table 6.1: Speedup for workflow environment with 2000 stocks and batch size 2000.

mechanism of the task farm architecture.

### 6.6 Summary

The design and implementation of the parallel workflow presented in this section offers some insights into the challenges of processing and analyzing high-frequency time series data. First, we proposed an architecture that allows the system to execute on a cluster of heterogeneous machines within
Chapter 6. The Parallel Workflow

<table>
<thead>
<tr>
<th>Processors</th>
<th>Response time statistics (seconds)</th>
<th>avg</th>
<th>med</th>
<th>stddev</th>
<th>var</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>14.49 14.55 0.69 0.48</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6.83 6.85 0.32 0.10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>3.74 3.73 0.23 0.054</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Response time statistics for varying processor configurations.

A wide-area network. The main reason for this is so the system can scale as the size and frequency of the input data increases. While a system designed for shared-memory machines may have a lower communication overhead, the scalability of such a system is constrained by the hardware limitations.

A major challenge was deciding how to, if at all, homogenize the independent streams of high-frequency data. There is a vast array of research dedicated to time series analysis, with many different approaches for dealing with missing or duplicate data samples. For simplicity, we transform inhomogeneous time series into homogeneous ones through interpolation and aggregation. The process is performed on the streaming data online and in real-time. In fact, a strict requirement of our system from the start was that all the processing in the pipeline should be performed online and in real-time. While this may seem trivial for simple tasks such as filtering, robust correlation and clique-finding are computationally-intensive and thus are the obvious candidates for a bottle-neck in the overall system response time. To determine the scalability of the system as a whole, we ran the system using an increasing number of processors for these two computationally-intensive stages. To get a sense of the workload requirements of a real-life stock exchange, we fixed the problem size at 2000 variables, which is roughly the number of securities traded on the Toronto Stock Exchange. As shown in Section 6.5, the system scales linearly up to 36 processors (18 for each of the robust correlation and clique-finding stages). Average response times of 14.49, 6.83 and 3.74 seconds were achieved using 5, 10 and 18 processors respectively. Interpolating from these results, we believe that our workflow could process a 2000 variable data stream in under a second using approximately 128 processors for the entire system. The final requirement of our system was that it should be flexible enough to support a variety of user-defined configurations. Our solution to this problem was the control process that communicates with the leaders of each stage, allowing for the dynamic adjustment of control parameters during the execution of the system.

The system described in this chapter is a working prototype, and the
implementation of it has been an iterative process. Like any other piece of software, there is always room for improvements. One challenge we have encountered thus far is how to efficiently transmit the output to the end-user. Each time new data enters the system, the updated results must somehow be disseminated to the end-user or application. While many application-specific design considerations must be taken into account, the key factor common to all is the speed at which the results can be sent over the network. For example, if the target application is high-speed automated trading, then the results must be summarized/compressed in such a way that they can be transmitted with minimum response time. If the data is being used in a human-centered visualization system, then a delayed response time of a second or two might be acceptable, allowing for extra time to provide more detailed results. Our original approach to transmitting the data to the client was to package the results in XML and send through a standard TCP/IP socket connection. Since then, it has become clear that for real-time transmissions, a more clever encoding scheme must be used to minimize the data size of the results. Potential future work in this area involves designing and implementing a data delivery protocol and format that satisfies the aforementioned latency requirements.

Another issue worthy of discussion is with respect to the adaptive message polling mechanism described in Section 6.4.4. Currently, all OPPLS processes are synchronized with the same poll_interval value after every graph update. In our original experiments this approach worked well since the cluster machines were all the same speed and had minimal work loads. However, when the cluster is composed of heterogeneous machines with varying work loads, enforcing processes to synchronize the value of their poll_interval parameter prohibits the system from performing load-balancing in the OPPLS component. This is because processes on slower machines will take longer to execute the poll_interval search steps, while processes on faster machines will execute the same number of search steps much quicker, but subsequently be forced to wait for the slower processes to finish before re-synchronizing the poll_interval parameter. Instead, we have realized that allowing each search process to independently update its own poll_interval parameter is a better solution since it allows for load-balancing amongst the OPPLS search processes.
Chapter 7

A Real-time Stock Market Application

In this chapter we investigate a direct application of the system proposed in Chapter 6 targeted towards the financial industry. The goal of our system is to perform real-time correlation analysis and clustering of high-frequency intra-day trading data. The output from the system can potentially be used by pre-trade analytic tools for automated/algorithmic trading or knowledge discovery applications in high-throughput electronic exchanges.

The stock market is a dynamic system, composed of multiple heterogeneous agents acting to maximize his/her own utility [3]. Within this complex financial network, researchers are finding subtle hints of emergent complexity and non-random patterns [9, 14]. Special "momentum" indicators previously known to exist on physical stock market floors are emerging in digital form [103]. LeBaron suggests that patterns within financial time-series span multiple time scales, and are somehow connected to the emergent behaviour of investors [104]. Intelligent automated trading agents and algorithms have been designed to exploit minor price inefficiencies [4, 80, 83, 105]. With decision response times being a critical factor in maximizing returns, the rush is on to design online and adaptive algorithms powered by high-performance parallel computing frameworks [4, 106]. All this suggest that, to some extent, the market is not as efficient as once believed, and that predictable, short-term patterns may be present within high-frequency data.

This work builds on a long list of recent research that examines stock market networks as complex systems [10, 11, 12, 13, 14, 107]. In most cases, edges represent the cross-correlation of the two stocks' log daily price return over a given time period (usually several years). Our approach considers the dynamics involved with the intra-day evolution of the market graph. The topology of this graph, which we call the dynamic intra-day market graph, is a snapshot of the system at a specific moment — it evolves over time. At any point in time, potentially useful information can be extracted from the dynamic intra-day market graph, and further computational analysis can be employed to find clusters or "cliques" of stocks that exhibit highly
correlated trading patterns.

While market graph studies involving long-term data focus on price-price interaction patterns, our method utilizes newly available high-frequency intra-day data to construct a dynamic intra-day market graph whose edge weights correspond to the correlation coefficients between pairs of evenly spaced technical indicator time series. These higher-frequency time series represent alternative views of the underlying market microstructure, from which potentially economical information can be extracted [14; 34, 79, 108]. Similar to existing approaches, we are looking for groups of stocks exhibiting highly correlated trading activity, but rather than restricting our correlation analysis to stock returns, we study the correlation dynamics of the technical indicator time series.

7.1 Technical Analysis

Technical analysis is a form of statistical analysis of financial data that assumes that past price and volume provide indicators of future price movements [109]. Unfortunately, there is much controversy over the usefulness of technical analysis, due to the theoretical results of the Efficient Market Hypothesis (EMH). The EMH states that a market is efficient with respect to information set \( \Omega \), if it is impossible to make economic profits by trading on the basis of information set \( \Omega \). While there are several forms of this hypothesis, each one representing a different set of information \( \Omega \), all reduce to suggest that when new information about a stock is released to the public, the share price already has that information factored into the price, and thus there is no profit-making opportunities using technical analysis. However, as trading data has become increasingly available, several studies have shown that there are indeed anomalies within the market [110, 111, 112, 113], suggesting that perhaps the market is not perfectly efficient as some claim. The subject of EMH, however, is an ongoing debate, and thus out of scope for this thesis.

In this chapter, we explore whether there may be useful information in correlating time series of evenly spaced technical indicator values. While the standard approach for correlation in finance is to use the time series of daily log returns, we are exploring a new realm of correlation analysis on the assumption that the technical indicators hold intrinsic value with respect to the price of the underlying financial instrument.

It is important to note that the particular indicators we have chosen in this study are not particularly significant. However, some technical indica-
tors do not produce a numeric value; they only result in discrete answers like "buy/sell", or "yes/no". Thus the only condition we impose when choosing a technical indicators is that it must produce a numerical value, i.e., a continuous signal from which we can periodically sample.

7.2 Stock Market Time Series Data

Financial time series are broadly divided into two categories: long-term historical price time series and short-term intra-day time series. Long-term price time series involves sampling the daily closing price of a stock over a span of many years; usually, the longer the time period the better, as it results in more robust models less susceptible to over-fitting of a particular short-term trend. Short-term intra-day time series can be further sub-divided into two types of time series: trades and quotes. An intra-day trade time series records the actual trades (and hence current stock price), while a quote time series represents the sequence of best bid and ask prices for a given stock. The bid price is the highest price someone is willing to pay for a stock, and the ask price is the lowest price someone is willing to sell a stock. The average of the bid and ask is called the midpoint or midprice, while the difference is called the spread. The following subsections further discuss these two types of financial time series.

The data we used in this study is Trade and Quote (TAQ) data from the Toronto Stock Exchange (TSX). The specific data set we investigate is from March 2005, and contains 1920 stocks with approximately 200,000 trades and 1,000,000 quotes per day. The main reason we chose this data was because it was easily accessible, although we were also keen on encouraging research on Canadian markets. Unfortunately, however, the TSX is a small exchange in comparison to other exchanges, for example the NASDAQ, which have significantly higher volume of both trades and quotes. Therefore, while the analysis undertaken in this thesis uses data from the TSX, our system is built to handle larger exchanges with much higher frequency and volume of trades and quotes.

7.2.1 Historical Data

When analyzing long-term financial time series data, a commonly used metric is the log return, which for stock $s_i$ is calculated as
Chapter 7. A Real-time Stock Market Application

$$R_i = \log P_i(t) - \log P_i(t-1) = \log \frac{P_i(t)}{P_i(t-1)}$$

where $P_i(t)$ is the closing price of stock $s_i$ at time $t$ (where time is usually measured in days). This metric has been applied to daily return data in various market graph studies [10, 11, 12, 13], resulting in a network model showing historical price-price interactions. More specifically, the equal-time correlation coefficient $\rho_{ij}$ between the returns $R_i$ and $R_j$ of stocks $s_i$ and $s_j$ is calculated as

$$\rho_{ij} = \frac{\langle R_i R_j \rangle - \langle R_i \rangle \langle R_j \rangle}{\sqrt{\langle R_i^2 \rangle - \langle R_i \rangle^2} \sqrt{\langle R_j^2 \rangle - \langle R_j \rangle^2}}$$

where $\langle R_i \rangle$ and $\langle R_j \rangle$ denote the average of the return vectors $R_i$ and $R_j$, respectively, over the given time period; e.g., if the time period under consideration is $N$ days, then $\langle R_i \rangle = \frac{1}{N} \sum_{i=1}^{N} R_i(t)$. Such a model is often used by portfolio managers to maximize diversity (and hence reduce risk) and to ensure proper allocation of total portfolio funds to various classes/sectors of assets. Unfortunately, this approach only works with historical data, and a different metric must be used to analyze intra-day data, since it would result in only a single sample. Even if $R_i$ was evaluated using $P_i(t)$ as the price after each trade, the number of trades within a day for most stocks are too few to justify a statistically significant correlation. One solution is to use a modified version of this metric which utilizes intra-day quote data, which is higher frequency data and is a good approximation of the true price of the asset [3].

7.2.2 Intra-day Data

High-frequency, intra-day quote data is the finest resolution of financial data available. Only recently has such data become easily accessible to the public. The research team of Olsen et al. have spent the last decade studying such high-frequency (primarily currency) data [3].

Table 7.1 shows sample quote data from the Toronto Stock Exchange. This example illustrates how there can be multiple quotes every second, and sometimes multiple quotes per symbol every second.

Using Equation 6.1 from the previous chapter, we can analogously par-
Chapter 7. A Real-time Stock Market Application

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Symbol</th>
<th>Bid Price</th>
<th>Ask Price</th>
<th>Bid Size</th>
<th>Ask Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>13:23:02</td>
<td>AE.DB</td>
<td>161.506</td>
<td>173.804</td>
<td>454</td>
<td>104</td>
</tr>
<tr>
<td>13:23:02</td>
<td>AVN.DB</td>
<td>145.202</td>
<td>158.8</td>
<td>450</td>
<td>150</td>
</tr>
<tr>
<td>13:23:02</td>
<td>AY.DB</td>
<td>113.9</td>
<td>113.603</td>
<td>113</td>
<td>503</td>
</tr>
<tr>
<td>13:23:02</td>
<td>G</td>
<td>22.708</td>
<td>23.008</td>
<td>128</td>
<td>98</td>
</tr>
<tr>
<td>13:23:02</td>
<td>G</td>
<td>22.708</td>
<td>23.008</td>
<td>138</td>
<td>98</td>
</tr>
<tr>
<td>13:23:02</td>
<td>IAI.SV.A</td>
<td>40.404</td>
<td>36.905</td>
<td>95</td>
<td>35</td>
</tr>
<tr>
<td>13:23:02</td>
<td>RIM</td>
<td>103.305</td>
<td>104.505</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>13:23:02</td>
<td>RIM</td>
<td>103.305</td>
<td>104.505</td>
<td>75</td>
<td>65</td>
</tr>
<tr>
<td>13:23:02</td>
<td>RIM</td>
<td>103.305</td>
<td>104.505</td>
<td>65</td>
<td>75</td>
</tr>
<tr>
<td>13:23:02</td>
<td>RIM</td>
<td>103.305</td>
<td>104.505</td>
<td>135</td>
<td>75</td>
</tr>
<tr>
<td>13:23:02</td>
<td>RIM</td>
<td>103.305</td>
<td>104.505</td>
<td>135</td>
<td>65</td>
</tr>
<tr>
<td>13:23:02</td>
<td>SJR.NV.B</td>
<td>29.304</td>
<td>30.004</td>
<td>44</td>
<td>14</td>
</tr>
<tr>
<td>13:23:02</td>
<td>SW</td>
<td>18.1</td>
<td>18.2</td>
<td>80</td>
<td>90</td>
</tr>
<tr>
<td>13:23:02</td>
<td>TRP</td>
<td>37.509</td>
<td>38.009</td>
<td>429</td>
<td>829</td>
</tr>
<tr>
<td>13:23:02</td>
<td>WRM</td>
<td>9.904</td>
<td>0.004</td>
<td>2464</td>
<td>454</td>
</tr>
</tbody>
</table>

Table 7.1: One second of sample quote data from the TSX.

Table 7.1: One second of sample quote data from the TSX.

In order for us to use the quote data within our calculations, we derive a new time series using a combination of the midpoint and open-high-low-close (OHLC) formats, as introduced in [82]. The midpoint format is simply the midpoint of the bid and ask price, e.g., \( \frac{1}{2}(\text{bid} + \text{ask}) \). The OHLC format creates new data samples, often referred to as bars, by aggregating multiple quotes within a given time window. The OHLC format is formally described next.

The set of ticks \( Q_i \) for stock \( s_i \) is converted to a set \( \Theta_i \) of OHLC tuples using a time window of \( \Delta t \) seconds. After aggregation, the data can be...
Chapter 7. A Real-time Stock Market Application

expressed as the set:

$$\Theta_t = \{(o_k, h_k, l_k, c_k, b_k) \mid 0 < k \leq L; k \in \mathbb{Z}^+\}$$

where $L = \left\lfloor \frac{T}{\Delta t} \right\rfloor$ is the number of bars when using time window $\Delta t$, $T$ is the total number of seconds in a trading day, and $o_k, h_k, l_k, c_k$ denote, respectively, the open, high, low and close midpoint price for bar $b_k$. Each tuple in $\Theta_t$ by definition aggregates multiple ticks within the given time window into a single OHLC data sample. For example, if $\Delta t = 5$, then each OHLC sample specifies the open, high, low and close midpoint values over the quotes within the corresponding 5-second time window.

Due to the sporadic nature of the data, a more rigorous definition of $o_k, h_k, l_k$ and $c_k$ is required to handle special cases (e.g., extended periods of time with no data); for details we refer to the literature [83].

7.2.3 Working With Multiple Time Series

Because of the sporadic nature of the data, a naive sampling of the time series data for multiple stocks will result in sample points which are not evenly aligned in time. See Figure 7.1 for an example of this problem with two stocks using a sampling period of $\Delta t = 1$ second. At time $t = 5$, stock A will have 4 data samples but stock B will have only 2. In this thesis we concentrate mainly on homogenous time series, although we make a preliminary investigation into the usefulness of examining inhomogeneous time series.

Homogenous Time Series

Most correlation-based analysis methods require the time series to be homogenized using an interpolation scheme. In order to produce multiple homogeneous time series, we force each sample to include a technical indicator
value for every stock. We do this by converting quote data to OHLC data points as described above, and complement with previous-tic interpolation to fill in for missing samples if no new quotes arrive within the time interval. As previously mentioned, the process of aggregation via OHLC formatting was discussed in [83] for a single time series. Similarly, previous-tic interpolation is a standard method for homogenizing financial time series with missing data [3]. To our knowledge, the culmination of these two ideas, along with the use of a technical indicator (rather than using the raw trade or quote time series), is a novel approach.

There is an obvious trade-off in the frequency at which data is sampled. Sampling very frequently, e.g., every second or two, brings the analysis closer to 'real time'. The problem, however, is that for most stocks, the frequency of quote updates is much slower than this sampling rate, and when interpolation is applied, the result is many consecutive constant values, giving rise to numerical stability issues in the correlation computation. On the other hand, if we increase the sampling window, for example to 30 or 60 seconds, the number of constant values decrease, but the result response time also increases. A particular challenge in this respect is choosing a sampling frequency which works well for as many stocks as possible. Table 7.2 shows quote statistics for a normal day of TSX trading using varying time windows.9

\[9\]We determined 'normal' by calculating averages for volume, price volatility and number of quotes over the entire month (March 2005), and then chose the day showing the least deviation from these averages (March 22).

<table>
<thead>
<tr>
<th></th>
<th>1 sec</th>
<th>2 sec</th>
<th>3 sec</th>
<th>4 sec</th>
<th>5 sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>max</td>
<td>157</td>
<td>240</td>
<td>351</td>
<td>442</td>
<td>510</td>
</tr>
<tr>
<td>avg</td>
<td>24.42</td>
<td>49.75</td>
<td>75.08</td>
<td>100.41</td>
<td>125.75</td>
</tr>
<tr>
<td>med</td>
<td>24</td>
<td>51</td>
<td>80</td>
<td>109</td>
<td>138</td>
</tr>
<tr>
<td>Q(0.9)</td>
<td>39</td>
<td>81</td>
<td>121</td>
<td>161</td>
<td>203</td>
</tr>
<tr>
<td>Q(0.1)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>stddev</td>
<td>22.27</td>
<td>43.22</td>
<td>63.76</td>
<td>84.05</td>
<td>104.25</td>
</tr>
<tr>
<td>variance</td>
<td>496.14</td>
<td>1867.61</td>
<td>4065.93</td>
<td>7065.19</td>
<td>10867.88</td>
</tr>
<tr>
<td>varcoeff</td>
<td>0.91</td>
<td>0.87</td>
<td>0.85</td>
<td>0.84</td>
<td>0.83</td>
</tr>
</tbody>
</table>

Table 7.2: Quote frequency distributions from our sample TSX quote data. Q(0.9) and Q(0.1) are the 0.9 and 0.1 quantiles, respectively.
Chapter 7. A Real-time Stock Market Application

Inhomogeneous Time Series

The issue of dealing with inhomogeneous, high-frequency stock market data is an open problem [4, 114, 115]. The usual approach is to homogenize the data samples through interpolation and aggregation before performing the correlation analysis. Recent work suggests that the temporal irregularity of the time series should be considered a feature of the system, rather than a problem which needs to be eliminated through interpolation or similar smoothing mechanisms [4]. Others believe that the irregular time series may exhibit patterns spanning multiple time scales, and are somehow connected to the emergent behaviour of investors [104]. Other recent approaches to dealing with inhomogeneous financial time series include Fast Fourier transforms [114] and covolatility-weighted sampling to adjust for data frequency differentials [3].

It has been shown that the simpler methods such as the interpolation and aggregation make inherent assumptions which may not hold in practice, and can introduce data bias into the analysis [3]. For this reason, and by extrapolating from the several lines of experimental work described above, we believe there is useful information to be discovered by analyzing the correlation behaviour between multiple, inhomogeneous time series. For example, when considering homogenous time series data, the patterns under investigation are assumed to be correlated in time. But what if there exist correlated patterns across a spectrum of time scales? This is the question we wish to address by treating the presence of new data samples (e.g., quotes) as events; that is, we do not synchronize data elements over time. Instead, we sample the data stream at regular intervals, but do not use interpolation to generate new data points in the presence of missing data. We do however, retain the OHLC format to compress multiple data points within the time window into a single data point. By taking this approach, we are able to detect correlated patterns across different time scales. Another advantage is that only a subset of the correlations need to be recalculated each time. This approach, however, also presents challenges of its own, raising questions such as “How can such patterns be interpreted?”, or “What does it mean for these events to be correlated?”.

Because high-frequency stock market data analysis is a relatively new concept, there are no obvious answers to these questions. By providing a framework such as the one proposed in this thesis, we offer a novel tool for investors and market analysts to interactively explore this new and interesting information space.
7.3 Technical Indicators as a Similarity Measure

Deciding on an appropriate measure for correlation is a difficult problem which depends on many factors, such as the time scale on which patterns are sought, how and by whom the results are to be interpreted, and the number and variety of stocks included in the analysis. Different similarity functions will produce vastly different market graphs. As discussed in Section 7.2.1, previous market graph research uses daily closing prices to model long-term price-price interactions using the log return metric. Thus, to construct the intra-day market graph, different data and metrics must be used. Fortunately, the rapid advancement of electronically-driven stock markets has enabled access to both historical and real-time high-frequency intra-day data. Using this newly available data, we construct the intra-day market graph where the edge weights are the correlation coefficients of the time series produced by calculating a technical indicator at evenly spaced time intervals. These higher-frequency time series represent alternative views of the underlying market microstructure, from which potentially economical information can be extracted [14, 34, 79, 108].

Our approach is partially motivated by Dempster et al., who use genetic programming to construct an optimal portfolio of short-term technical indicators for use within their automated trading system [82, 83, 84] While they use several advanced indicators for trend identification and movement prediction, for simplicity we experimented with three basic indicators: the Quote Midprice Rate of Change (QMROC), Quote Frequency Simple Moving Average (QFSMA) and the Quote Volume Simple Moving Average (QVSMA). The quote midprice is a close approximation of the true price, while other measurements such as the spread, quote frequency and arrival times are all closely linked with liquidity and volatility. We now formally define the three technical indicators used in our study.

The Quote Midprice Rate of Change (QMROC) indicator at bar \( m \) using a \( n \)-period time window, where \( 0 < n \leq m \), is calculated as:

\[
QMROC(m, n) = \left(\frac{c_m - c_{m-n}}{c_{m-n}}\right) \times 100
\]

where \( c_i \) is the closing price at bar \( i \).

The Quote Frequency Simple Moving Average (QFSMA) indicator is meant to measure the level of "activity" of a stock, and is defined similarly to the price simple moving average, but rather than using the closing price at each bar, we use the number of quote updates. Formally, the Quote Frequency Simple Moving Average at bar \( m \), where \( 0 < n \leq m \), is defined as:
Chapter 7. A Real-time Stock Market Application

$$QFSMA(m,n) = \frac{1}{n} \sum_{i=0}^{n-1} a_{m-i}$$

where $a_i$ is the number of quotes at bar $i$, and is defined as

$$a_i = \{q \in Q | timestamp(q) \in [t_i - \Delta t, t_i]\}$$

Finally, we define $QFSMA(m,1) = a_m$.

We also investigate the dynamics of the quote volume (size of bid/ask quotes) by calculating the Quote Volume Simple Moving Average (QVSMA). The calculation is defined almost identically to the QFSMA above, except rather than counting the number of quotes in each time window, we count the sizes of each quote bid and ask. Formally, the Quote Volume Simple Moving Average at bar $m$, where $0 < n \leq m$, is defined as:

$$QVSMA(m,n) = \frac{1}{n} \sum_{i=0}^{n-1} v_{m-i}$$

where $v_i$ is the sum of the quote bid and ask sizes at bar $i$, and is defined as

$$v_i = \sum_{j=1}^{M} v_{j}^{bid} + v_{j}^{ask}$$

where $v_{j}^{bid}$ and $v_{j}^{ask}$ are the sizes (volumes) of the $j$-th bid and ask quotes respectively, and $M$ is the number of quotes received during the last time interval, i.e.,

$$M = \{q \in Q | timestamp(q) \in [t_i - \Delta t, t_i]\}$$

Again, we define $QVSMA(m,1) = v_m$.

7.4 A Parallel Workflow for High-Frequency Stock Market Data

This section describes how we tailor the generic parallel workflow system described in Chapter 6 to handle real-time, high-frequency intra-day stock market data. Figure 7.2 illustrates the specialized system components.

The Data Collector currently pulls quotes from a database or file. This component can be replaced by one which connects to a live data feed over the Internet without affecting the rest of the pipeline. The Filter & Accumulator batches up quotes within the specified time window (every $\Delta t$ seconds) and calculates the new technical indicator values for each stock.
The technical indicator values are then passed as the new set of data samples to the **Robust Correlation** component, which updates its internal data sample queue, dropping older samples if necessary. Robust Correlation then recalculates the correlation coefficients and passes these values as edge weights to the **Clique-based Clustering** component, which updates its internal graph data structure as necessary. Finally, after searching for cliques for a given time (dependent on data input frequency), the cliques, representing clusters of highly correlated stocks, are sent to the **Results Analysis Server** which transmits the information to a remote client application on an external network.

The main difference between the application-specific instantiation shown here in Figure 7.2, and the generic process structure shown in Figure 6.1, is the process by which input data is collected and processed before sending to the Robust Correlation and Clique-based Clustering stages. The generic workflow generalizes the input data to a set of time series streams sampled discretely at a regular interval. In this instantiation, the input data is a stream of quote data containing time series for all stocks in the TSX, and the variables to be correlated (the data samples) are the technical indicators calculated after every sampling period. The correlation and clique-finding stages work the same for both instantiations. The last stage of the pipeline (Results Analysis Server) is also customized to this stock market application in the sense that the output has specific meaning; e.g., a subset of stocks showing correlated behaviour of some technical

![Figure 7.2: An instance of the parallel workflow for real-time stock market analysis.](image-url)
indicator over a given time window.

### 7.4.1 Computing Environment

Our current testing environment is comprised of two small compute clusters. The first contains 14 PCs, each equipped with dual Intel Xeon 3.06GHz CPUs with 1GB of cache, 2GB of RAM, running SuSe Linux 9.1 with kernel version 2.6.5-7.252-smp. The second cluster contains 6 PCs, each equipped with dual Intel Xeon 2.0GHz CPUs with 512KB of cache, 4GB of RAM, also running SuSe Linux 9.1 with kernel version 2.6.5-7.252-smp. While this seems to be sufficient for handling the TSX data with minimal response time, the system would undoubtedly need to scale several orders of magnitude to process data from larger exchanges in real-time. Just to provide some comparison, the TSX contains just under 2,000 stocks with approximately 1 million quotes and 200,000 trades recorded per day. INET [95], an electronic exchange hub which integrates several other exchanges, offers direct access to over 7,200 US exchange-listed securities, with approximately 65 million quotes and 5.7 million trades per day — a significantly higher volume and frequency in comparison to the TSX.

### 7.4.2 Correlation Calculation

Correlation analysis is widely used in finance for portfolio optimization, derivatives pricing, risk management and pairs/spread trading [114]. The correlation calculation is performed exactly as described in Section 6.3.1, except now variables are stocks, and data samples are technical indicators. Each time a new quote is received for stock \( s_i \), its correlations between all other stocks needs to be computed. Using Equation 6.4, we can calculate \( \zeta^t \), the exact number of correlation values which need updating at time \( t \). This value will vary depending on whether the system is using asynchronous or synchronous data sampling. If synchronous data sampling is used, then at each bar there will be exactly one new data sample for each variable, and so \( \zeta^t = \frac{N^2 - N}{2} \), where \( N \) is the total number of stocks in the system.\(^{10}\)

Otherwise, the value of \( \zeta^t \) represents only those correlation coefficients which were affected by the new data samples.

\(^{10}\)As discussed in Section 6.2, synchronous sampling often requires data interpolation and/or aggregation to homogenize the time series.
Chapter 7. A Real-time Stock Market Application

7.4.3 Maintaining the Dynamic Intra-day Market Graph

In order to create a network model representation of our data, a similarity measure needs to be defined. As discussed in Section 7.4.2, our system maintains the real-time $N \times N$ correlation matrix for a set of $N$ stocks. We define the similarity between stocks $s_i$ and $s_j$ to be $\rho_{ij}$, their correlation coefficient.

\[
similarity(s_i, s_j) = \rho_{ij}, \text{ where } \rho_{ij} \in [-1,1] \text{ for } 1 < i, j < N, \ i \neq j
\]

The exact variables used in the correlation analysis have been discussed in detail in Section 7.3. Then for each pair of stocks $s_i$ and $s_j$ in the network, we connect them with an edge with weight $\similarity(s_i, s_j)$. Table 7.3 shows a hypothetical correlation matrix for five stocks.

<table>
<thead>
<tr>
<th></th>
<th>SUNW</th>
<th>RHAT</th>
<th>INTC</th>
<th>GOOG</th>
<th>ADBE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNW</td>
<td>1</td>
<td>0.05</td>
<td>0.87</td>
<td>0.60</td>
<td>0.27</td>
</tr>
<tr>
<td>RHAT</td>
<td>1</td>
<td>0.42</td>
<td>-0.35</td>
<td>0.95</td>
<td></td>
</tr>
<tr>
<td>INTC</td>
<td>1</td>
<td>0.66</td>
<td>0.75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOOG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-0.59</td>
</tr>
<tr>
<td>ADBE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Table 7.3: Hypothetical correlation matrix for five stocks.

After the correlation matrix is calculated, the graph is then constructed and thresholded as shown in Figure 7.3.

7.4.4 Clique-based Clustering

As described previously, we use OPPLS to find not only the maximum clique, but also as many maximal cliques as possible. Because there is an exponential number of maximal cliques, we currently set a maximum limit on the number of maximal cliques which OPPLS stores at any given time.\(^{11}\) This way, we approach the problem of clustering by finding a large set of maximal cliques from which we can extract the most tightly connected (and thus correlated) subsets of stocks. The current implementation of OPPLS includes only one minor modification which addresses a specific characteristic.

\(^{11}\)In our experiments we use a limit of 5000-10000, although this value is somewhat arbitrary and may be changed to suit other constraints (e.g., RAM).
of the stock market graph. In general, one is interested in only sufficiently high correlations, and thus the resulting market graph can be expected to be relatively sparse. Section 5.2 describes the modification which improves the performance of OPPLS on sparse graphs. The use of a parallel and online algorithm is key for extracting clusters in the rapidly changing market graph. Because even the smallest time delay can potentially result in significant profits (or losses), it is important to utilize every possible means of speedup in order to exploit time-sensitive opportunities.

7.4.5 A Prototype Visualization Client

In this section we present our solution to the market graph visualization problem. First we describe how nodes and edges are drawn, and then we describe how we use a force-directed layout algorithm to provide additional visual indicators of clustering results. Figure 7.4 shows a typical closeup view of a small area of the market graph, displaying only positively correlated
stocks passing the user-defined threshold.

Developing our tool was greatly simplified by prefuse[116], a powerful Java-based toolkit for interactive visualization of graphs. Our approach is essentially another application of Jeff Heer's recent Vizster [117] project, which is a visualization tool for exploring online social networks. Social networks, along with many other real-world networks, exhibit the same scale-free properties [107]. Scale-free networks are a class of network models where some vertices are adjacent to many edges, thus acting as "hubs", while most vertices have a low degree. Furthermore, a scale-free network implies that the network characteristics (such as the one just mentioned) are independent of the number of vertices in the network. We implemented several of the same visualization techniques with slight modifications to suit our application, as it has been shown that financial networks also exhibit small-world properties [13]. We note here that our approach to visualizing

![Figure 7.4: Snapshot of a market graph with positive correlations.](image)

the output is only one of many possibilities. Depending on how the output of the system is to be used, different visualization methods may be desired. It may also be the case that no visualization of the output is required; e.g.,
Chapter 7. A Real-time Stock Market Application

the output of our system is used as direct input into an automated trading system.

**Force-Directed Layout**

To effectively visualize the subgraphs of the market graph, we use a force-directed layout algorithm. Briefly, a force-directed layout algorithm abstracts edges into springs and vertices into electrically charged particles. The graph then represents a physical system, whereby “forces” are applied to the particles (vertices) causing them to expand or contract under the tension of the springs. The algorithm proceeds iteratively until an equilibrium state is reached. For simplicity, we use the default force-directed algorithm provided by the Prefuse toolkit, and tweak the edge length and spring coefficient functions to optimize the layout, such that highly related clusters appear close together, while independent clusters and minimally-related clusters are placed further apart.

**Interaction and Information Integration**

We utilized many of the built-in features from the Prefuse toolkit such as geometric pan and zoom, overview display for retaining global context, and animations for interpolating the color and size of display objects during transitions between states. One advantage to working with stock market data is that information is readily available from a variety of sources. Additional information for each stock can be accessed via a context menu which supports on-demand retrieval of live news feeds for real-time stock news, aggregated from various sources using RSS technology. Other context menu options provide convenient access to a Google search or Yahoo Finance quote page for the stock under consideration.

**Visualizing the Dynamic Market Graph**

As discussed earlier, a major goal of this system was to support the visualization of real-time market data. Thus, we have designed our system so that it is capable of performing dynamic graph layout as updates are received in real time. Upon startup, the visualization client connects to the Results Analysis Server, the data output stage of the pipeline. Clique results are passed to the Results Analysis Server which are then transmitted via a standard TCP socket connection to the client, which then updates the visualization display accordingly.
Chapter 7. A Real-time Stock Market Application

Analyzing Clique Overlap

Since the state of the market market is constantly evolving, the set of computed cliques is also constantly changing. We therefore need a method that can appropriately handle dynamic clique sets and produce meaningful output. One such approach is based on analyzing the extent of clique overlap in the market graph. Two specific methods for analyzing clique overlap are the co-clique matrix (CCM) approach and the k-clique graph approach.

A co-clique matrix (CCM) measures for each pair of stocks the number of times they appeared together in a clique. Specifically, each entry \((i,j)\) in a co-clique matrix specifies how many times stock \(s_i\) appeared in a clique with stock \(s_j\). Recent research has shown that the overlapping clique structure within real-world networks (e.g., collaboration, word-association and protein interaction networks) exhibit non-trivial correlations and small-world scaling properties [102]. Because our system produces a series of dynamic market graphs, we also dynamically maintain the co-clique matrix; thus, we are simultaneously tracking and analyzing the evolution of the underlying clique overlap structure. Formally, a co-clique matrix \(CCM(t)\) for \(N\) stocks is a \(N \times N\) matrix where \(CCM(t)(i,j) = |\{c \in C(t) | i, j \in c\}|\), where \(C(t)\) is the set of all maximal cliques discovered by OPPLS during the time interval \([t - \Delta t, t]\). Furthermore, the matrix diagonal \((i,i)\) specifies the number of cliques containing stock \(s_i\), which can also be considered a measure of \(s_i\)’s centrality [101]. Stocks with high clique overlap are of particular interest because removing them tends to dramatically change the market graph and corresponding clique overlap structure.

A k-clique graph \(G(t)(k)\) is a graph that contains only edges whose corresponding co-clique value at time \(t\) is greater than \(k\); that is, \(G(t)(k) = (V,E(k))\), where \(V\) is the set of all stocks in the underlying graph \(G(t) = (V,E)\), and \(E(k) = \{(i,j) \in E | CCM(t)(i,j) > k, i \neq j\}\). For example, when \(k = 1\), then \(G(t)(1)\) is simply the graph containing all the vertices and edges from the set of maximal cliques found during the time interval \([t - \Delta t, t]\).

In the context of financial networks, both of these approaches have not been previously explored and thus we take a first step at investigating the nature of overlapping clique structures within the market graph.

7.5 Evaluation

In this section, we present some anecdotal results obtained by running our system on real-world market data. The current system was tested on a dataset consisting of the Trade and Quote (TAQ) data from the Toronto
Chapter 7. A Real-time Stock Market Application

Stock Exchange (TSX) for March 2005; additionally, we performed some tests on synthetic data to ensure the correctness of the techniques. Furthermore, our investigation is the first (to our knowledge) which attempts to study the evolving intra-day market graph. We stress here that our goal is not to show ground-breaking discoveries on short-term correlation patterns, but rather to demonstrate how basic clique information representing correlated sets of stocks can be extracted from the dynamic intra-day market graph.

7.5.1 Homogeneous Time Series

Interpreting homogeneous time series correlations is a relative easy task. Determining the cause of the correlation is another question altogether (and hence out of the scope of this thesis). As discussed in Section 7.3, applying standard time series analysis to high-frequency data is problematic as it results in many constant values. For example, we found that correlating our indicators using a sampling frequency of less than 10 seconds often resulted in many consecutive constant values, which gave rise to an undefined correlation coefficient (since standard deviations of zero occur). Our original hypothesis was that using high-frequency data would eliminate this problem; we see now that while the frequency of the input data has increased, much of this data is essentially repetitive, offering little or no change to the underlying time series. As a result, we have a small but representative example of the type of cliques we wish to extract from the intra-day market graph. For this analysis we did not perform any back-testing over longer time durations. Instead, we just extracted a single day of trading data which we calculated as a 'normal' day on the TSX.\textsuperscript{12} Because the TSX data we currently have is relatively low-frequency, in our experiments we use a longer sampling interval to reduce the number of interpolations required to account for missing data. This problem will be much less pronounced for higher-frequency data from exchanges such as the NASDAQ or NYSE.

Figures 7.5, 7.6 and 7.7 are results showing correlated QFSMA indicators. Figure 7.5 shows three energy stocks (two Encana stocks and Talisman), and CP Holdrstm, a holdings company for Canadian Pacific Railway, which have highly correlated QFSMA indicators using a 3 minute sampling period and a sample size of 50. We can see that the two Encana stocks have nearly identical curves, while Talisman and CP Holdrstm follow suit with

\textsuperscript{12}We did this by calculating averages for volume, price volatility and number of quotes over the entire month (March 2005), and then chose the day showing the least deviation from those averages (March 22).
similarly correlated curves. The average pair-wise correlation over these stocks as computed by the Maronna method is $\rho = 0.89$.

Figure 7.5: Energy stocks showing correlated QFSMA indicators with 3 minute sampling interval.

Figures 7.6 and 7.7 are taken from the same experiment, which shows correlated QFSMA indicators using a 1 minute sampling period with a sample queue size of 100. Figure 7.6 shows three energy trust funds, whose QFSMA indicators exhibit a spike at approximately the same time. This sudden flurry of activity could represent, for instance, an important news release. Detecting such spikes in activity would be a valuable tool for momentum traders who rely on the ability to identify short-term movements. Figure 7.7 shows four stocks (AberDiamond, Advantage Energy Trust, Cameco and Gerdau Ameristeel) that are somewhat related in the sense that they are in a resource or related industry, as well as one non-related stock (Cognos). The pair-wise correlation co-efficients for these time series are all between 0.9 and 1.0. While at first glance these values may seem surprisingly high, the robustness of Maronna essentially down-weights outlier samples that would otherwise decrease the correlation. Their QFSMA charts are somewhat opposite to the previous chart in the sense that the indicator values appear highly correlated for some time, and then they all suddenly drop to a very weak signal. This type of activity could represent the situation when all the
stocks were simultaneously correcting (reversing a recent gain/loss surge).

Figure 7.8 shows stocks which exhibit correlated QVSMA indicators using a 30 second sampling period. The QVSMA indicator we use combines both bid/ask volumes, and so it represents a general liquidity indicator. Calculating the QVSMA indicator using only one of the bid or ask volumes would provide an indication of current buy or sell liquidity, respectively. The resulting time series for Cambior and PlacerDome (both gold mining companies) are positively correlated with $\rho = 0.84$. TLC Vision is also found to be positively correlated with Cambior ($\rho = 0.95$) and PlacerDome ($\rho = 0.80$), although the magnitude of change in the indicator series is noticeably less (which may indicate a spurious result due to noise). Weston Foods, on the other hand, exhibits a QVSMA indicator series which is highly negatively correlated to all three stocks, with an average correlation co-efficient of $\rho = -0.82$.

Figure 7.9 shows three stocks (Agrium, Cott Corp, Russel Metals) exhibiting correlated QFSMA indicators using a 3 second sampling period. Figure 7.10 shows scatter plots of the indicator values for each pair of stocks over the same time interval. Each point in the plot represents the indicator value for the two stocks at that moment in time. The average correlation
7.5.2 Inhomogeneous Time Series

Analyzing correlated behaviour from inhomogeneous time series is difficult because of the potential variations in frequencies between the time series. For example, consider the extreme case where two inhomogeneous time series containing 100 data samples each span non-overlapping time windows — one series may represent data between the interval 9:00-9:05, and the other may represent the time interval 10:15-14:00. While our initial intuition rejects the validity of correlating these time series, we believe there may be useful information to be extracted from time series which are inhomogeneous, but with a less extreme temporal shift as the case presented above. That is, we wish to consider inhomogeneous time series with partially overlapping time windows and approximately equal durations. Furthermore, depending on the degree of time window overlap, the results may represent a time-lagged correlation; e.g., the case where the time windows spanned by the two inhomogeneous time series are approximately the same size and one is shifted forward or backwards in time. How exactly to extract meaningful economic information from such results is another question, and answering it is bet-
Chapter 7. A Real-time Stock Market Application

Figure 7.11 shows three stocks exhibiting correlated event-based patterns using the QMROC indicator. The average correlation coefficient of these time series is $\rho = 0.52$; while not convincingly high, a quick glance at the chart shows a clear co-movement in the indicator values. The three stocks are Otelco Inc (OTTO-UN.TO), a provider of wireline telephone services, Rogers Communications (RCI-B.TO), and Spectrum Signals (SSY.TO), software developer for defense electronics applications. Figure 7.12 shows the same data points when they are shown in their correct time series order. It is difficult to discern a relationship between those same stocks.

7.6 Potential Applications

This section discusses several potential applications of our system.

The first application of our system is to serve as a data exploration and knowledge discovery tool. Using advanced visualization and interaction techniques, end-users can navigate and explore through the evolving market graph in order to gain a deeper understanding of the complex underlying...
Chapter 7. A Real-time Stock Market Application

Figure 7.10: Three pair-wise correlation scatter plots for the stocks in Figure 7.9. The first company name corresponds to the values on the X-axis, the second name to the Y-axis.

correlation-based dependencies.

The second application is that of a decision support and recommendation system, which would provide valuable on-demand information using a combination of historical and real-time information from the market network model. Our vision for this model works as follows: A day trader makes a trade decision on a single stock using their preferred strategy (e.g., some combination of technical indicators and rule-based strategies).
Chapter 7. A Real-time Stock Market Application

Figure 7.11: Correlated event-based QMROC patterns.

Figure 7.12: The underlying time series for the QMROC events in Figure 7.11.
Chapter 7. A Real-time Stock Market Application

day trader could then query our system to find stocks to which it has highly correlated short-term patterns. The trader would then manually assess the recommendations to find additional trading opportunities.

The third application of our system is geared towards automated trading strategies, and in particular, automated pairs trading. Pairs trading involves finding a pair of stocks (in general, any pair of financial instruments can be used) which are known to be highly correlated, and when their price ratio diverges past a critical point, the two stocks are simultaneously bought long and shorted. The recent increase in access to historical and real-time trading data is leading to improved pairs trading models [119, 120, 121]. As pairs trading relies heavily on correlation analysis, the ability to compute a (near) real-time market-wide correlation analysis may offer new opportunities to perform accurate, real-time pairs trading for all possible pairs of stocks.

7.7 Summary

In this chapter we have proposed the use of a highly parallel workflow environment for correlating and clustering data from thousands of stocks in (near) real-time. As shown in Chapter 6, our system can process a data stream of 2000 time series with an average response time of 3.74 seconds. Our proposed method of correlating and clustering short-term indicators is a novel concept, but there are clearly many issues which need to be further addressed. For example, some of the results presented in Section 7.5 show correlated behaviour between stocks which, at least intuitively, appear to have no clear connection between them. These results are accurate in the sense that the indicator values form time series that are actually correlated, and the system correctly identifies them; however, many patterns of this nature may be spurious. It is well-known that common, market-wide factors (e.g., interest rates, foreign affairs, etc) contribute to some degree to the correlated behaviour. One way to address this issue is to divide the indicator value into its common and private components [3]. The common component value reflects information from market-wide factors and is common across all stocks (at least in a particular industry), whereas the private

13"Buying long" is what most people are familiar with — shares are bought at or near the current price, and later sold for a profit (or loss); the positions are reversed when the price ratio converges to some pre-determined level [118]. "Shorting" a stock means selling a stock first (technically, the shares are borrowed from a bank or similar institution), until a later time when the shorted shares can be bought back for a profit (or loss).
component value reflects the information specific to the stock. Correlating the private component value would result in a measure which excludes the common information, potentially leading to correlations which can more precisely predict co-movements.

Furthermore, we have found through a trial-and-error approach that the correlation of indicators on such short-term time scales are highly sensitive to the sampling period as well as to the queue length (number of samples used in the correlation). For example, indicators based on rate of change require that the queue size and sampling period match the time window in which the change is expected, otherwise sudden surges of activity will not be detected since the robust correlation calculation will treat those data samples as outliers. If the time window spanned by data samples in queue can closely approximate the time window of the activity, then the spikes in the indicator chart will span a larger portion of the time window, and thus the spikes will not be seen as outliers. On the other hand, detecting trends like moving averages requires a longer sampling period, otherwise the indicators tend to have a flat signal (no change), which results in an uninformative positive correlation.

Another interesting and open question is how to analyze the cliques discovered in multiple consecutive market graphs. Previous research on clique overlap [26, 101, 102] have focused on static graphs. To our knowledge, there is no existing work which examines how the clique overlap structure evolves within a dynamic graph. The ability to precisely characterize and quantify the temporal evolution of clique overlap structures within the dynamic intra-day market graph may yield valuable information about short-term movements.
Chapter 8

Conclusion

In the investing world, timing is key. When information pertaining to a stock is released, investors react, and the share price is adjusted accordingly. As described in this thesis, stocks can be modeled as a network, where network connections define some relationship or similarity measure between stocks. With each passing second of the day, new information is disseminated through various communication mediums, ultimately perturbing the current state of the network. Thus, the topology of the network is continuously evolving. A high-performance compute infrastructure with scalable, robust algorithms is essential in order to capture meaningful and economical information from this dynamic network of stocks. Two specific ways of exploiting information in this data is through correlation and clustering. Furthermore, as we have shown in this thesis, correlation analysis can be used to create a network model of a stock market using a variety of user-defined similarity measures. Then, further information can be extracted from the network model using a clustering technique such as clique-finding to identify subsets of stocks exhibiting highly correlated similarity measures. Unfortunately, due to the complexity of the computations involved, correlating and clustering thousands of variables in real-time is an extremely challenging task, and even minor accuracy errors or incorrect decisions could result in significant financial losses. In light of all this, we see a clear path to addressing this challenging problem. Thanks to advances in high-speed Internet links, affordable commodity clusters and multi-core chips, we believe the solutions lies in exploiting sophisticated parallel and online algorithms.

Throughout this thesis we demonstrate how a stochastic local search algorithm is inherently well-suited to handle a large-scale, time-sensitive combinatorial optimization problem such as clustering. We introduce the sequential, offline versions of DLS-MC and PLS, two state-of-the-art SLS algorithms designed for the maximum clique problem, and show how they can be adapted and employed within a fully distributed parallel workflow environment. First, we developed parallel PLS and parallel DLS-MC, which through the basic yet powerful multiple independent runs parallelization
strategy, are able to achieve impressing speedup results over a wide range of
problem instances. Next, we armed parallel PLS with trajectory continuation, a
technique which enables it to operate in a dynamic environment where
the input graph is subject to continuous modifications. The final result is
online parallel Phased Local Search, which is a scalable and adaptive SLS
algorithm capable of finding high-quality cliques within a dynamic graph.
We also presented preliminary results using a cooperative search strategy,
which although on average was out-performed by its non-cooperative vari-
ant, was dominant by several orders of magnitude for a particularly hard
problem instance. This suggests that the modifications introduced in the
cooperative variant may offer some clues on how to improve the original
PLS algorithm.

We then integrated online parallel PLS into a parallel workflow environ-
ment where it operates on a dynamic graph with an objective to find as
many maximal cliques as possible, under extremely tight time constraints.
The dynamic graph is constructed from the real-time correlation matrix of a
larger number of variables within a high-frequency time series data stream.
The correlation matrix is dynamically maintained using an online parallel
implementation of Maronna, a powerful correlation method for dynamic data
streams, robust to outliers and noisy data — a key features for dealing with
sporadic, highly irregular time-series. We also introduced several other im-
portant but non-parallel data processing components, and describe how they
fit within our data processing pipeline. In the same chapter we presented
timing results using synthetically generated data streams, and showed that
by using 36 processors we were able to process a high-frequency data stream
with 2000 input variables using time windows as small as 3.74 seconds.

In the final chapter, we performed a preliminary investigation into the
applicability of our system for processing high-frequency intra-day stock
market data in order to determine clusters of stocks exhibiting highly cor-
related time series of short-term technical indicator values. Our novel ap-
proach emphasizes a real-time market-wide analysis, and the flexible system
design enables easy interchanging of computational components, creating
a powerful framework for designing, testing and evaluating high-frequency
financial applications. With potential applications ranging from automated
trading systems to exploratory knowledge discovery, we believe our system
can empower its users with a deeper insight into the complex underlying
network structure of the stock market.
Future Work

The design and implementation of this system has been an iterative process. As with any research which is actively on the frontier of innovation, we were forced to make initial assumptions — many of which were revisited — to reduce the scope of the design space. The need for high-performance computing in the finance industry is clear; the question is how to best approach the problem. In this thesis we have presented a system which we believe is on the right track to meeting the computationally demanding requirements outlined above. The system is loosely coupled, with modularity in mind, meaning we consider the current configuration of computing components one instance of the parallel workflow environment. For example, OPPLS is just one approach to finding highly correlated subsets of stocks in the market graph. We could easily swap OPPLS for a different graph-based clustering algorithm, or, alternatively, the entire market graph model could be removed, leaving only the dynamic correlation matrix which could then be used as input into a different clustering algorithm.

In general, this thesis has explored the design space of a system for real-time analysis of high-frequency data streams. More importantly, though, this research has exposed many interesting and open problems in regards to high-performance computing within the context of high-frequency finance. The following section discusses many exciting extensions to our work.

Alternative Clustering Approaches

Our current clustering method is only one of many possible choices. Clustering is perhaps one of the most difficult data mining tasks as it requires a notion of similarity a priori, making the results susceptible to user bias. In particular, our method of clique-based clustering, while extremely relevant in some contexts, could be replaced by other graph-based clustering methods. For example, clustering by maximum-weight clique partitioning would avoid the problem of specifying a threshold value from which to accept or reject edges. Similarly, we could extract quasi-cliques instead of maximal cliques from the dynamic graph. A quasi-clique is a highly dense, but not necessarily complete, subgraph. By allowing quasi-cliques instead of maximal cliques, we could provide a more robust set of potential solutions. Furthermore, we believe it may be interesting to allow multiple clustering methods to be executed on the same dynamic graph, producing a set of (possible different) clustering results.
Chapter 8. Conclusion

Correlating Multiple Metrics for Multi-objective Optimization

It has been shown that automated trading of technical indicators in isolation are profit-losing strategies. However, when multiple technical indicators are used simultaneously, profitable trading strategies emerge. We are therefore keen on utilizing our massively parallel system to synchronously correlated multiple variables for each stock. Several approaches for dealing with multiple objectives exist. One solution is to employ a multi-objective clustering algorithm that can optimize over the wide range of (possibly weighted) variables. Multi-objective Variable Neighbourhood Search (MOVND) is a novel Stochastic Local Search algorithm for clustering weighted graphs using multiple scoring functions. The solution space forms a Pareto front of high-quality solutions, which could be explored by an interactive user in order to find an optimal portfolio weighting.

Correlating Time-Series on Multiple Time Scales

Currently, our system samples the data stream every $\Delta t$ seconds. If we sample evenly across all stocks every $\Delta t$ seconds, and retain $Q$ data samples in our data queue, we end up with a homogenous time series (assuming we used interpolation/aggregation to homogenize where necessary) spanning a time window of $Q\Delta t$ seconds. This means the correlations must exist within that time window. But what happens if we don’t know exactly what time window in which such correlated behaviour occurs? Or what if we want to compare correlations across multiple time scales? In this case, the solution would be to compute correlations on multiple time scales. For example, after the first 60 seconds (i.e., $t = 60$), there are 60 data points with $\Delta t = 1$, 30 data points with $\Delta t = 2$, 15 data points $\Delta t = 4$, 5 data points with $\Delta t = 12$ and 1 data point with $\Delta t = 60$. At time $t = 120$, the number of data points for each time interval is doubled, as well as a single data point with $\Delta t = 120$. The idea of finding related patterns across multiple time scales is discussed in a recent paper by LeBaron [104].

Time-Lagged Correlations

Our correlation method currently performs correlation on time series which span identical time windows. Calculating time-lagged correlations could provide useful information with regards to predicting future price movements. Our current implementation of online parallel Maronna could be modified to calculate a time-lagged correlation simply by using using data
samples from a time-shifted window. For example, consider the data sets \( \{d_{t+Q}, d_{t-Q-1}, \ldots, d_t\} \) and \( \{d_{jt-Q}, d_{jt-Q+1}, \ldots, d_{jt}\} \) which represent the \( Q \) most recent data samples for stocks \( s_i \) and \( s_j \), respectively, at time \( t \). Assuming \( \Delta t = 1 \) second, then a lagged correlation with lag of 1 time unit would correlate \( \{d_{t-Q}, d_{t-Q+1}, \ldots, d_t\} \) with \( \{d_{jt-Q-1}, d_{jt-Q}, \ldots, d_{jt-1}\} \), or vice-versa.

Dealing With Inhomogeneous Time Series

While the problem of inhomogeneous time-series is a known problem in the area of statistical analysis and data-mining, until recently it has not been studied in the context of high-frequency, short-term stock market data. Recently it was shown that a standard correlation calculation can be modified to take into account the frequency differentials in the data between two time series [3]. The modification requires only minimal additional computation and could be easily integrated into the Maronna method.

Clustering Data Streams

There is a large body of work dealing with data stream clustering [122, 123, 124]. Data stream clustering is the problem of how to maintain a consistently good clustering of a sequence of data points. One of the main differences between our clustering approach and data stream clustering approaches is that the data streams they consider are not online. Instead, the data streams represent data sets which are far too big to fit into main memory, making random access very expensive, and thus necessitating the need for single-pass scans through the data. More recently, however, there has been a dramatic increase in online data streams, which has spurred the development of novel online data stream clustering approaches [125, 126, 127]. While this thesis addresses the problem of online data stream clustering by finding cliques in a dynamic graph, we believe that future work in this area could involve applying online data stream clustering methods to the raw data streams before they are transformed and correlated.

Exploiting Additional Data

The high-frequency financial data stream used in this thesis (the bid-and-ask data) contains more data fields than we currently use. Our calculations only use the bid/ask price, bid/ask volume, and the time stamp for each data point. Each data point contains additional information, such as a buyer and seller ID, unique identifiers for each entity that submits orders into the
electronic market system. Also, there is potentially valuable information to be extracted from the depth-of-market data (also known as the limit order book), which is all bid/ask quotes other than the current best. This data is particularly useful in ultra short-term trading because it shows such things as how many people are buy and selling and at what price, the liquidity of the stock and the potential short-term direction of the stock.

Clique Coverage

While the original PLS algorithm is optimized for finding the maximum clique, OPPLS was designed to find a large set of maximal cliques. The "clique coverage" ability of a heuristic algorithm refers to the time and efficiency for approximating the full set of maximal cliques. Future work in this area would include a rigorous empirical analysis to accurately quantify the clique coverage performance of OPPLS. While this thesis did report any results on this issue, preliminary (but undocumented) experiments showed promising results. In order to measure the clique coverage performance of OPPLS, we had to record all unique maximal cliques that were discovered during the search. Obviously, with \(3^\frac{n}{2}\) potential maximum cliques, some care must be taken to efficiently store and compare the maximal cliques. To implement this idea, we borrowed a technique used in the Reactive Search Framework [58] for detecting cycles in the search trajectory using an incremental hash function. We use the same hash function to generate a unique clique identification number, and store that instead of the clique vertices. Since we can compute the hash value in \(O(n)\) time, checking the uniqueness between existing and newly discovered maximal cliques is \(O(|K|)\), where \(|K|\) is the size of the current maximal clique. As suspected, preliminary results from these experiments showed OPPLS was able to find a large percentage of clique with large cardinality, but a relatively small number of maximal cliques with low cardinality.
Bibliography


Bibliography


Bibliography


Appendix A

Scalability and Speedup
Results for PPLS, PDLS-MC
and CPPLS

This appendix provides full results on the scalability and speedup experiments from Chapter 4.
Figure A.1: Scalability and Speedup results for PPLS on the brock800.1 instance. Plot (a) shows for varying numbers of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length.
Figure A.2: Scalability and Speedup results for PPLS on the p.hat1500-1 instance. Plot (a) shows for varying numbers of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length.
Appendix A. Scalability and Speedup Results for PPLS, PDLSTM and CPPLS

Figure A.3: Scalability and Speedup results for PPLS on the C1000.9 instance. Plot (a) shows for varying numbers of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length.
Appendix A. Scalability and Speedup Results for PPLS, PDLs-MC and CPPLS

Figure A.4: Scalability and Speedup results for PPLS on the keller6 instance. Plot (a) shows for varying numbers of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length.
Figure A.5: Scalability and Speedup results for PDLS-MC on the brock800.1 instance. Plot (a) shows for varying number of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length.
Figure A.6: Scalability and Speedup results for PDLS-MC on the p.hat1500-1 instance. Plot (a) shows for varying number of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length.
Figure A.7: Scalability and Speedup results for PDLS-MC on the C1000.9 instance. Plot (a) shows for varying number of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length.
Appendix A. Scalability and Speedup Results for PPLS, PDLS-MC and CPPLS

Figure A.8: Scalability and Speedup results for PDLS-MC on the keller6 instance. Plot (a) shows for varying number of processors the RLDs while plot (b) shows the corresponding speedup based on median run-length.
Appendix A. Scalability and Speedup Results for PPLS, PDLS-MC and CPPLS

Figure A.9: Scalability and Speedup results for CPPLS on the brock800.1 instance. Plot (a) shows speedup based on the median run-length, while plot (b) shows Speedup based on the median run-time.
Figure A.10: Scalability and Speedup results for CPPLS on the p_hat1500-1 instance. Plot (a) shows speedup based on the median run-length, while plot (b) shows Speedup based on the median run-time.
Figure A.11: Scalability and Speedup results for CPPLS on the C1000.9 instance. Plot (a) shows speedup based on the median run-length, while plot (b) shows Speedup based on the median run-time.
Appendix A. Scalability and Speedup Results for PPLS, PDLs-MC and CPPLS

Figure A.12: Scalability and Speedup results for CPPLS on the keller6 instance. Plot (a) shows speedup based on the median run-length, while plot (b) shows Speedup based on the median run-time.
Appendix B

Performance Results for OPPLS

This appendix provides full results from the dynamic graph series experiments in Chapter 5.
Appendix B. Performance Results for OPPLS

Figure B.1: Online Performance for p.hat500-1 20-stage Additive Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Figure B.2: Online Performance for brock200.1 20-stage Additive Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Figure B.3: Online Performance for keller4 20-stage Additive Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Figure B.4: Online Performance for p.hat500-1 20-stage Subtractive Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Figure B.5: Online Performance for brock200.1 20-stage Subtractive Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Figure B.6: Online Performance for keller4 20-stage Subtractive Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Figure B.7: Online Performance for p.hat500-1 10-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Appendix B. Performance Results for OPPLS

Figure B.8: Online Performance for brock200.1 10-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Appendix B. Performance Results for OPPLS

Figure B.9: Online Performance for keller4 10-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.

175
Figure B.10: Online Performance for p.hat500-1 20-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Figure B.11: Online Performance for brock200.1 20-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Figure B.12: Online Performance for keller4 20-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Figure B.13: Online Performance for p.hat500-1 40-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Appendix B. Performance Results for OPPLS

Figure B.14: Online Performance for brock200J 40-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.

(a) brock200.1

(b) brock200.1 / Cumulative

Figure B.14: Online Performance for brock200.1 40-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Figure B.15: Online Performance for Keller4 40-stage Mixed Dynamic Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Figure B.16: Online Performance for the Additive Dynamic Market Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Appendix B. Performance Results for OPPLS

Figure B.17: Online Performance for the Subtractive Dynamic Market Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.
Figure B.18: Online Performance for the Mixed Dynamic Market Graph Series. Plot (a) shows the individual number of search steps needed to find the maximal clique at each stage in the series, while plot (b) shows the total cumulative selections.