Scalable clustering algorithms and optimization methods for parallel architectures

Andrei B. Khlopotine

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2015

Reading Committee:

Vikram Jandhyala, Chair

John Sahr

Aakash Tyagi

Program Authorized to Offer Degree:

Electrical Engineering

©Copyright 2015
Andrei B. Khlopotine

University of Washington

Abstract

Scalable clustering algorithms and optimization methods for parallel architectures

Andrei B. Khlopotine

Chair of the Supervisory Committee:
Professor Vikram Jandhyala
Electrical Engineering

Clustering algorithms provide a way to analyze and understand huge amount of data that is present and evolving today in various areas such as sciences, engineering, marketing, finance, etc. Although numerous serial clustering approaches have been developed, only few of them are viable nowadays given algorithm complexities and sizes of problems. The focus of this dissertation are the parallel optimization methods and tuning techniques that will enable the computing society to perform clustering of massive data on shared memory parallel architectures.

The first part of the dissertation investigates clustering methods and their parallel optimization for data represented by coordinates on a two-dimensional Cartesian plane. These clustering methods are based on the well-known plane sweep algorithm that scans a coordinate plane with a goal to find intersections of the rectangles. The applications of these algorithms find their places in earth surface processing, very large scale integration (VLSI) design and military surveillance, just to cite a few. The contribution presented in this dissertation is a

simple and highly scalable plane sweep algorithm named Scan-List (SL). Despite this method having a higher order of sequential operation complexity than other well-known serial algorithms, its scalability allows it to surpass those algorithms when run in parallel. This algorithm is profiled against the best-known plane sweep method and was applied to tests generated using industrial Electronic physical design automation (EDA) tools.

The second part illustrates clustering methods focused on data represented as network systems or graphs with node and edge sets. The practical applications of graph clustering algorithms are nowadays popular social networks, biological research, marketing and financial products. The methodology presented in this thesis investigates and filters only those clustering algorithms that produce acceptable qualities while being low complexity methods and are capable on working with massive data. A well-known Label propagation algorithm (LPA) is characterized to be a unique algorithm that meets the requirements and in addition is capable of efficient scaling on shared memory parallel architectures. The present deficiencies of LPA are addressed to achieve the linear scalability on a conventional shared memory Intel® Xeon architecture. Intel® Xeon Phi (Phi) Label Propagation algorithm (PLPA) is presented. PLPA is the first community detection algorithm implemented on a Phi that is a novel many integrated core architecture. PLPA is fully scalable up to 32 cores and achieves above 100 speedup on Phi with a maximum of 56 cores and 224 hardware threads while maintaining the quality of detected communities. The analysis as to why the speedup is limited by the Phi hardware, and how this shall be resolved in the next generation of MIC based products is provided. The existing possibilities to utilize Phi on massive networks that cannot fully fit in a limited capacity Phi memory are illustrated. A PLPA extension, a modified Phi based LP

algorithm PLPA-M to utilize Phi on a network with billions of edges is presented and the scalability analysis is provided. Future opportunities for massively parallel processing of networks using LPA on Phi (multiple cards) and other architectures are analyzed. We provide heuristics for hybrid LPA implementations that would enable LPA on next more advanced generation of heterogeneous Phi platforms and distributed types of architectures. Finally, we provide initial empirical work and set a base for future research.

To my Family and to my Colleagues

TABLE OF CONTENTS

List of Figures IX

List of Tables XI

Acknowledgments XII

Chapter 1: Introduction 1

Chapter 2 : A Variant of Parallel Plane Sweep Algorithm for Multi-Core Systems . 7

2.1 Motivation 7

2.2 Introduction 8

2.3 Algorithm 12

2.3.1 Problem definition 12

2.3.2 Naïve approach. 13

2.3.3 Fundamental approach. 14

2.3.3.1 Scan Priority Trees. 14

2.3.3.2 Scan Priority Search Tree algorithm 18

2.3.3 Proposed algorithm 21

2.3.4 Efficiency analysis and serial run results 25

2.4 Parallel implementation and results 27

2.4.1 Implementation 27

2.4.2 Experimental setup. 32

2.4.2 Results 37

Chapter 3 : Optimizing Parallel Label Propagation based Community Detection on the Intel® Xeon Phi™ Architecture 41
3.1 Motivation 41
3.2 Introduction 42
3.3 Related work 48
3.4 Label Propagation Algorithm 51
3.4.1 Definitions 51
3.4.2 Label Propagation Algorithm 53
3.5 Experimental setup 58
3.5.1 Intel® Xeon Phi™ 60
3.5.2 Networks and methods 60
3.6 Label Propagation on Phi 61
3.6.1 Initial optimization on Intel® Xeon series 61
3.6.2 Optimizing LPA on Phi 64
3.7 Label Propagation on Phi with massive networks 70
3.7.1 Introduction 70
3.7.2 Experimental results 72
3.8 Heterogeneous LPA and opportunities for massively parallel architectures. . 74
Chapter 4 : Conclusions and future work 80
Bibliography 82

List of Figures

Figure Number | Page

1.1 An example of a physical design. Intel P54C processor. 2
1.2 An example of a surface map. A partial map of Spokane, WA. 3
1.3 A 250 node random graph. 4
2.1 Iso-oriented rectangles. 10
2.2 Pseudo-code for BF solution. 14
2.3 Pseudo-code for a rectangle insertion into PST. 15
2.4 Pseudo-code for rectangle enumeration using PST. 16
2.5 Pseudo-code for a rectangle removal from PST. 17
2.6 Scan Priority Search Tree (SP) algorithm applied to rectangles. 19
2.7 Pseudo-code for SP solution. 20
2.8 Scan-List algorithm visualized for rectangle R1. 22
2.9 Rectangle density histogram for the 7 M test with mean. 23
2.10 Pseudo-code for SL solution. 24
2.11 Serial run results comparison SP versus SL. 27
2.12 Load balancing issues with SP algorithm visualized. 28
2.13 SP dynamic workload balancing with work-stealing. 29
2.14 Pseudo-code for finding slice coordinates for SP static balancing. 30
2.15 Pseudo-code for parallel SL solution. 31
2.16 Differences of synthetic (left) and EDA (right) layouts. 33
2.17 Rectangle density histogram for a fragment of the 7 M dataset. 34
2.18 Rectangle density histogram for a fragment of the 13 M dataset. 34
2.19 Rectangle density histogram for a fragment of the 1 M synthetic dataset generated with initial settings of the generator. 35
2.20 Rectangle density histogram for a fragment of the 1 M synthetic dataset generated with initial settings of the generator. 36
2.21 Scaled SL vs scaled SP for the 7 M rectangles test. 39

2.22 Scaled SL vs scaled SP crossover trend line. 40

3.1 Example of sparse graph degree distribution. 44

3.2 Example of a small network with “hub” nodes. 45

3.3 A community detection process visualized. 46

3.4 Girvan-Newman edge betweenness based community detection algorithm visualized. 48

3.5 Louvain community detection algorithm visualized. 50

3.6 Common types of graphs. 52

3.7 An example of LPA processing a nine node network. 53

3.8 A low level node connectivity illustration example. 55

3.9 LPA pseudocode. 56

3.10 Optimized LPA pseudocode. 57

3.11 Intel® Xeon Phi™ high level platform architecture. 58

3.12 PLPA pseudocode. 65

3.13 Strong scaling of PLPA (x-axis, core count, base-2 log-scale) running time in seconds. 67

3.14 Intel® Xeon Phi™ high level platform architecture, the second generation. . . 70

3.15 Modularity measurements for PLPA versus serial LPA. 70

3.16 PLPA-M pseudocode. 72

3.17 Strong scaling of PLPA-M (x-axis, thread count, base-4 log-scale) running time in seconds. 73

3.18 Random partitioning that results in poor clustering and oscillation. 75

3.19 HLPA pseudocode. 76

3.20 Suboptimal utilization of available threads due to synchronization points. . . 78

3.21 Modularity measurements for HLPA versus base LPA. 78

List of Tables

Table Number | Page

2.1 SP running time breakdown, in seconds. 26

2.2 SL running time, in seconds. 26

2.3 Maximum deviations for each scaling experiment. 30

2.4 Scaled SP performance for 7 M case, in seconds. 37

2.5 Scaled SL performance, in seconds. 38

3.1 Intel® Xeon Phi™ platform specifications. 59

3.2 Graphs used in the experiments. 61

3.3 Xeon run results. Total serial LPA time and the first iteration scaling measurements. 63

3.4 PLPA results with SMT turned on modified networks. 69

3.5 HLPA time and the first iteration scaling measurements (in seconds) on Xeon. 79

Acknowledgements

Firstly, I would like to thank my adviser Professor Vikram Jandhyala for his support and thank him for guiding me through the doctoral program. Professor Jandhyala has been always open to discuss any matters and suggest how I would benefit from following a specific path in my research studies. I enjoyed working with him on my research plan that started with relatively simple and well-known problems. This approach helped me to learn basics of parallel computing. I enjoyed challenges in the second half of the program when Professor Jandhyala guided me to work on the problems with high demand in both academia and industry.

I thank Dr. Arun V. Sathanur, a former ACE lab Assistant director, for helping me with the infrastructure and organization matters and also for helping me with technical writing and interim advising.

I thank Professor Aakash Tyagi, Dr. Desmond Kirkpatrick and Dr. Steven Burns, my former colleagues at Intel Corporation, for their efforts in supporting my Ph.D. studies and also for helping me to receive funding and allowing me to take some time on-leave for my full time studies in 2010.

I thank Professor John Sahr, Professor Alberto Aliseda, Professor Aakash Tyagi and Dr. Arun Sathanur for joining my Ph.D. committee.

Finally, I thank my family for supporting me at the time when I joined the Ph.D. program on a full time basis and had to leave the job at Intel. Hopefully, I can be a good example for my children from this perspective as well.

Chapter 1

Introduction

Clustering algorithms find their applications in many areas such as biology, computer science, hardware design, finance and marketing, just to cite a few. Clustering is an abstract and common definition for methods that search for groups of densely connected or densely placed entities in initial datasets. Nowadays, datasets reach billions of entities, thereby requiring optimization of initially developed clustering methods in order to maintain a processing speed. There is a variety of ways to represent datasets and this dissertation illustrates clustering methods of datasets that are represented by graphs and by coordinates on a Cartesian plane.

Traditional ways to improve complexities of clustering algorithms have been approximation, incremental design and parallelization. While approximation and incremental design methods can be characterized as more or less traditional optimization techniques, parallelization started getting attention recently as its progress depended on hardware support. Thankfully, various semiconductor design companies followed the trend and started releasing many variants of parallel hardware platforms such as GPUs, multi core and many core platforms and distributed systems. The hardware platforms kept evolving along with the increased demand for more computing power. Nowadays, computing communities have access to many flavors of both shared and distributed memory systems. Notably,

parallelization is not trivial and the most efficient serial algorithms are not guaranteed to achieve optimal performances when adopted in a parallel manner.

In chapter 2, a variant of a plane sweep algorithm that is embarrassingly parallel and therefore easily scalable on multi-core machines and clusters, while exceeding the best-known parallel plane sweep algorithms on real world and synthetic tests, is presented along with the empirical evaluations with the best-known existing techniques. The plane sweep method has been a well-known technique to scan planes and executing a variety of operations with shape

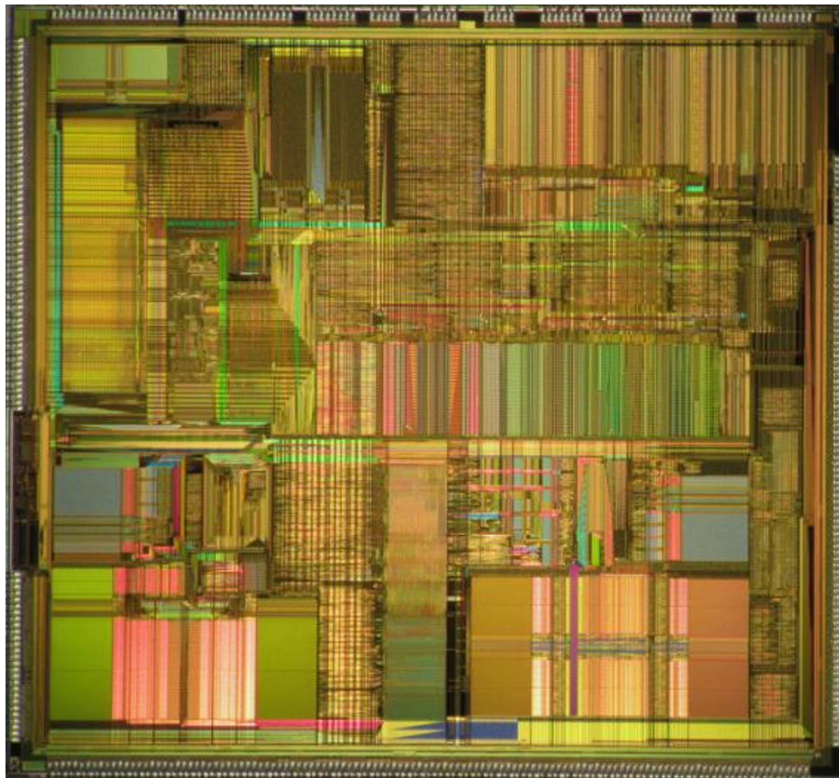

Figure 1.1: An example of a physical design. Intel P54C processor (Creative Commons License, Wikipedia)

intersections being one of them. The shapes are conventionally clustered and their surface views resemble sparse graph representations with data points represented by Cartesian coordinates. Examples of such datasets are present in VLSI design as back end layouts of custom ASIC blocks shown in Figure 1.1. Blocks of custom design logic such as caches and register arrays are highly clustered and have much higher than average densities of entities.

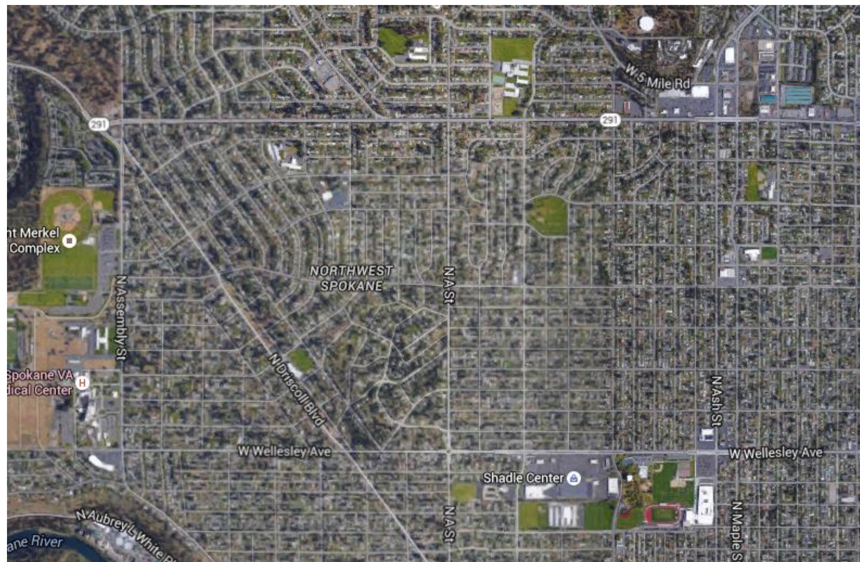

Figure 1.2: An example of a surface map. A partial map of Spokane, WA. The figure was generated using Google maps tool.

Other notable examples are earth surface maps with an example shown in Figure 1.2. These maps are used in online mapping, GPS navigation and military surveillance applications, just to cite a few. Similarly to the VLSI design example in Figure 1.1, some areas of a surface are clustered with higher densities of polygons. These areas can be residential or industrial

districts. At the same time forests and valleys are less populated and, therefore, are characterized by a relatively uniformal plane.

In both cases noted above, the high density clusters represent areas where the most computation occurs when processing the plane. Provided remaining areas on the plane have fewer polygons, this distribution pattern would cause disbalancing when processed in parallel, thereby preventing applications from scaling linearly. This dissertation provides research achievements of elaborating a parallel algorithm for the rectangle intersection problem that is a subset of a more generic polygon intersection problem. A variation of the plane sweep algorithm, the Scan-List algorithm (SL), is presented. The chapter provides analysis of the serial running time complexities and efficiencies of the best-known serial algorithms alongside the newly proposed SL technique. We also analyze parallel implementations of the naïve and conventional (fundamental) algorithms. All these parallel implementations use straightforward divide-and-conquer approaches for the rectangle intersection problem that are likely to scale well. Finally, we provide empirical results of running SL on large industrial and synthetic datasets applicable in ASIC design.

In chapter 3 we shift our attention to community detection in networked systems represented by sparse graphs. Typical examples of such graphs include social and molecular networks with node and edge counts reaching millions and billions entities respectively. An example of a small network is shown in Figure 1.3. Firstly, we analyze existing community detection algorithms with a purpose of choosing the algorithms that may benefit from parallelization techniques. We also illustrate various constraints and deficiencies such as complexities and performances.

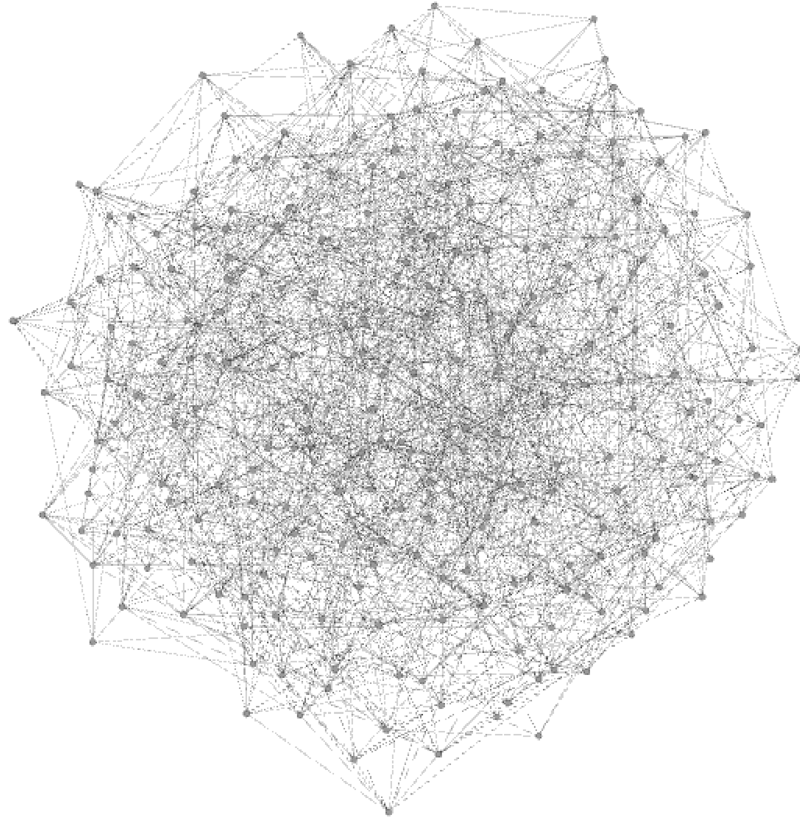

Figure 1.3: A 250 node random graph. The figure was generated using Gephi network analysis tool.

We introduce the fundamental serial label propagation algorithm (LPA) and its synchronous, asynchronous and semi-synchronous variations. LPA is known for its linear complexity and is capable of processing large graphs. We then briefly discuss the existing parallel LPA implementations and address the current deficiencies to tune LPA to run on a shared memory parallel architecture. We illustrate various datasets from the *Stanford Large Network Dataset Collection* (SNAP) [40] used in our experiments. We also summarize the

key features of our platform that includes a Phi coprocessor. We presents PLPA and empirical evaluations of running PLPA with the test networks. We also illustrate current platform limitations that prevent optimal PLPA scaling on Phi. Finally, we devise the ways to utilize Phi on massive networks that do not fit into Phi memory. We present PLPA-M and emphasize the enhancements it brings in for large scale community detection on Phi.

Finally, we present heuristics to apply LPA on heterogeneous and distributed types of architectures. We analyze several possibilities of refactoring base LPA and provide initial empirical work of running a semi-synchronous version of LPA on Xeon platform with a goal to understand scalability limits. This sets a base for future research for LPA based scalable community detection on massively parallel architectures.

Chapter 2

A Variant of Parallel Plane Sweep Algorithm for Multi-Core Systems

2.1 Motivation

The plane sweep problem finds its applications in GPS mapping services, military surveillance and, notably, in semiconductor designs as a critical component of VLSI computer aided design (CAD) tools. Physical CAD applications, such as the design rule checker (DRC) and parasitic extraction (RC), have been integral to the process of designing VLSI chips. Nowadays, custom chips contain billions of transistors. For example Intel® Phi coprocessor contains more than five billion of transistors. Each transistor physically translates into a higher order of rectangles that are present in the layout. For example, a simple inverter translates into tens of physical rectangles that needed be processed while designing an electric circuit. With increases in numbers of cells per a functional block into the millions, physical applications such as DRC and parasitic extraction for RC are becoming more and more computationally intensive. Nowadays, these applications must be capable of processing billions of entities while sweeping layout planes and can run dozens of minutes to hours per a design block. Having these applications run faster will not only improve overall product time to market but also enable engineers to be more effective since communication with applications this way

can be more interactive without running them in overnight mode. In general, parallelization has been one of the popular ways to optimize application performances. The emergence of affordable multi-core processors has enabled engineering community to access low cost shared memory systems. However, parallel algorithms used in custom chip physical design bring together challenges for their efficient and effective design and implementation. In addition, there is a limited number of publications and analysis available on parallel implementations and most of available work is deficient in such key features as implementation simplicity, robustness and scalability. A simple and easily programmed application would be the best candidate to investigate how to improve these aspects.

This dissertation focuses on parallel implementations of a rectangle intersection problem - a simple enough problem for our purpose. The rectangle intersection problem is a subset of the plane sweep problem a topic of computational geometry and a component in DRC and RC physical design applications.

2.2 Introduction and related work

The rectangle intersection is a computational geometry problem and is a subset of a generic intersection problem that deals with sets of planar objects such as line segments, circles, half-spaces, etc. This topic became of high interest in end of 1970s and the development continued into the 1980s. The most motivation for the research came from computational geometry studies in general and as far as rectangle intersection is concerned

from optimizing EDA such as DRC, RC extraction and masking flows. An accurate circuit representation in simulations is required to achieve a high product yield. DRC is an integral part of the ASIC design flow and makes sure that a physical design complies with the design and manufacturing rules for a particular manufacturing technology. Some basic DRC algorithms and flow is well described in [63]. DRC problem can be formulated as follows: given a separation required on a metal layer, expand top and right sides of all rectangles by that separation and then compute intersections. The computed intersections mean the corresponding rectangles are too close and violate design rules. Many physical design applications require a polygon view of design layout as input. For example, the basic design rules that are checked in DRC are width, spacing and enclosure and DRC tools need input in a form of rectilinear polygons. Likewise RC extraction also called parasitic extraction or parasitics is an application that deals with accurate representation of an analog model of the circuit. Good basics for parasitic extraction are provided in [64]. An extraction problem can be describe as computing intersections both in same layer and in layers above and below to compute "neighbors". Then lookup tables are used as a speedup over using field solvers. Parasitics include parasitic capacitances, resistances and inductances. Capacitance and resistance extraction tools require polygon view of design layout as inputs.

A rectangle is a subclass of polygons. The rectangle intersection problem is defined as finding the set containing all overlapping (intersecting) pairs of rectangles in a given set of N rectangles with parallel sides. For a non-strict case, intersection can include a common edge or in case of DRC be a specific spacing between two test rectangles. This dissertation deals

with two-dimensional iso-oriented rectangles (hereafter rectangles) whose sides are parallel to the coordinate axes [46] as shown in Figure 2.1.

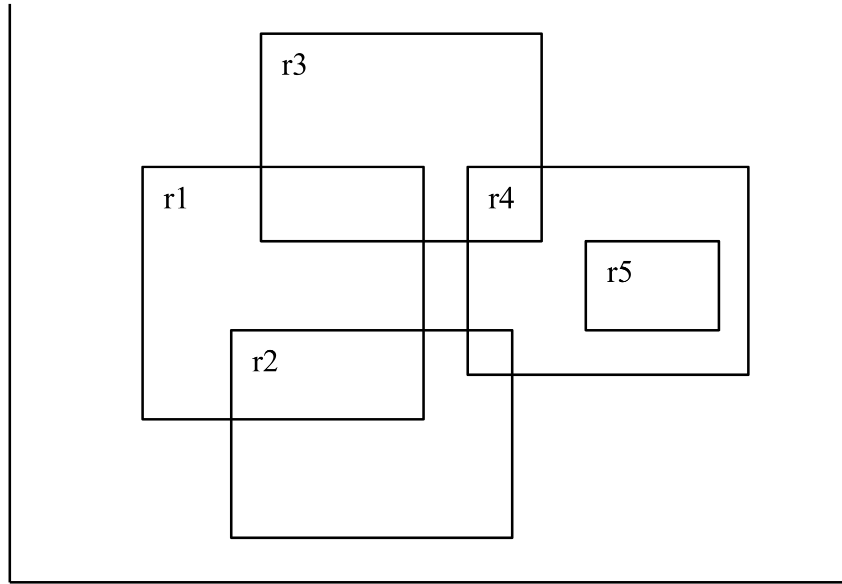

Figure 2.1: Iso-oriented rectangles.

The naïve brute force method (BF) consists of checking all rectangle pairs for intersection or containment. Its running time is $O(N^2 + K)$ with K being the number of intersecting pairs. This method has unacceptable runtimes with today's application-specific integrated circuit (ASIC) designs with millions of rectangles and for which parallel scalability will not be sufficient to make runtimes acceptable.

An algorithm for finding intersection of two of N general plane objects was presented in [43] while achieving $O(N \log N)$ running time. A fundamental plane sweep algorithm for

reporting and counting all intersecting pairs among N planar objects with $O(N \log N + K \log N)$ running time and $O(N+K)$ space with K being the number of intersecting pairs was proposed by Bentley [44].

The generic plane sweep algorithm was applied to rectangle intersection problem in [46, 50] achieving $O(N \log N + K)$ running time but suboptimal $O(N \log N)$ memory. The deterioration in the memory came from using complex data structures such as the segment tree that required additional memory footprint. The memory was reduced to $O(N)$ by Edelsbrunner [45] using so-called rectangle trees. An alternative solution to obtain linear space usage was proposed by McCreight [47] with introduction of priority search trees (PST). In parallel with the plane sweep algorithm, a divide and conquer approach with doubly linked list structures was developed by Lee [48] while also achieving optimal $O(N \log N + K)$ running time and $O(N)$ space.

The search for parallel solutions for calculating rectangle intersections started several years ago and now several mature solutions such as [54-57] are available. In the earliest work, Chow [58] presented a solution to the rectangle intersection problem. McKenney *et al.* [55] presented implementation based on [46]. Batista *et al.* [54] came up with the solution based on [58]. These solutions achieve sublinear speed-ups with number of cores and therefore are deficient in the key feature of scalability.

We make the observation that each improvement in serial sweep algorithms also has additional complexity. This complexity, some of which is due to the use of heuristics, may be difficult or even impossible to parallelize efficiently and therefore limits overall algorithm scalability. Therefore, we take the approach of devising a simple plane sweep algorithm with

minimum complexity and with high potential for scalability. Our proposed algorithm is extremely practical and easy to program in serial and in parallel while being inherently balanced, achieving optimal scalability and outperforming straightforward parallel implementation of the best-known plane sweep methods. We do not consider fine-grain locking of any of the sweep-line data structures as this is not straightforward and is unlikely to scale well. A second contribution of this research is the development of parallelizable static balancing heuristics for a conventional plane sweep method.

2.3 Algorithm

2.3.1 Problem definition

We represent a rectangle R in the following way: *R.leftX / R.leftY* and *R.rightX / R.rightY* for lower left and upper right rectangle corners respectively. Given two rectangles *R1* and *R2*, formulas (1, 2) can be used to define the conditions under which rectangles *R1* and *R2* intersect or have a common edge. These simple formulas are similar to the ones presented in [58] and can be easily modified to include a parameter for spacing analysis applications such as DRC.

$$R1.leftX <= R2.rightX \;\; AND \;\; R1.rightX >= R2.leftX \qquad (2.1)$$

$$R1.leftY <= R2.rightY \;\; AND \;\; R1.rightY >= R2.leftY \qquad (2.2)$$

Without loss of generality we use the open interval definition above for intersection. Once it is determined that the two rectangles do intersect, formulas (3-6) are used to compute the intersection rectangle.

$$intersectionR.leftX = max\ (R1.leftX\ ,\ R2.leftX) \quad (2.3)$$

$$intersectionR.leftY = max\ (R1.leftY,\ R2.leftY) \quad (2.4)$$

$$intersectionR.rightX = min\ (R1.rightX,\ R2.rightX) \quad (2.5)$$

$$intersectionR.rightY = min\ (R1.rightY,\ R2.rightY) \quad (2.6)$$

We make the reasonable assumption that input rectangles stored in EDA database are, without loss of generality, sorted by *leftX* coordinate to facilitate EDA generic operations such as queries and are in processor memory at the beginning of operation.

2.3.2 Naïve approach

The simplest algorithm to solve a rectangle intersection problem is to check each pair of rectangles if their coordinates overlap. The pseudo-code for this naïve approach is shown in Figure 2.2. Provided the complexity of this approach is $O(N^2 + K)$, this solution can be used as a verification engine on smaller datasets.

```
    Input: rectangle list rSet
    Output: rOutputSet
1   for ∀r1 ∈ rSet do
2     for ∀r2 ∈ rSet do
3       if r1 and r2 intersect using (2.1, 2.2) then
4         compute intersection rectangle using (2.3, 2.4, 2.5, 2.6);
5         add intersection rectangle to rOutputSet
6       end if
7     end for
8   end for
```

Figure 2.2: Pseudo-code for BF solution.

2.3.3 Fundamental approach

2.3.3.1 Priority search trees

There can be various options for the data structure that holds active rectangles. However, the data structure must be capable of supporting several key operations such as rectangle insertion, rectangle deletion and rectangle intersection checks against active rectangles. The last operation is called rectangle enumeration. Two most popular approaches to provide these operations are priority search trees and segment trees [47]. These two approaches provide the required operations with an equal complexity of $O(\log N)$. We choose priority search trees over segment trees since the latter require slightly higher memory footprint.

A priority search tree was introduced in [57]. These type of trees were developed in response to needs for multi-dimensional searching that supports range queries. Although each point on the plane has two coordinates, insertion, deletion and search operations on a tree

happen over a single dimension. Provided we sweep the plane along *X* dimension, *Y* is the coordinate that we use for the range queries.

```
    Input: rectangle A, minY, maxY
    Output:
1   if root empty then
2      add A to root
3      return
4   end if
5   node=root
6   while(true)
7      if A.leftY < node.leftY then
8         A is superior - swap node and A
9      end if
10     update mean to tempMid=(minY+maxY)/2
11     if A.rightY < tempMid then
12        if left subtree is empty then
13           insert A here
14           break
15        end if
16        update maxY=tempMid
17        traverse left side of tree
18     else
19        if right subtree is empty then
20           insert A here
21           break
22        end if
23        update minY=tempMid
24        traverse right side of tree
25     end if
26  end while
```

Figure 2.3: Pseudo-code for a rectangle insertion into PST.

Insertion, enumeration and deletion of rectangles are not trivial operations since there are two coordinates but a priority search tree (PST) is a single dimensional structure. As already

mentioned a PST was adopted for a rectangle intersection problem and adopted some significant changes to support all PST operations.

Once a rectangle becomes active, upon hitting its left edge, it is added to the tree and stays there until its right edge is encountered. To add a rectangle both *Y* values of the interval *[LeftY, RightY]* are used. The pseudo code for insertion is illustrated in Figure 2.3. Although there are two *Y* based coordinates, only *leftY* is independent and is used to look up the objects in a priority tree. The other coordinate *rightY* is also used but it does so in an increasing order only.

```
   Input: test, root, minY, maxY
   Output:
1  if root not empty and root.rightY < test.rightY then
2     if root.leftY > test.leftY then
3        found an intersection
4     end if
4     if test.leftY < (minY+maxY)/2 then
5        recurse left subtree
6     end if
7     recurse right subtree
8  end if
```

Figure 2.4: Pseudo-code for rectangle enumeration using PST.

Hence, contrary to conventional binary trees, PST resembles a 1.5-dimension tree. *MinY* and *MaxY* are the minimum and maximum for *Y* for all rectangles on the plane and represent a range for future queries. These values are known at the time when a problem is parsed. A tree is traversed while finding the insertion spot for the rectangle. Once the insertion spot is found, the swap of the nodes will happen and the existing node that provided an insertion spot will have to be inserted somewhere in the tree by traversing it.

```
    Input: rectangle A, minY, maxY
    Output:
1   node=root
2   while A is not removed
3      if node == A then
4         if left subtree exists then
5            if right subtree exists then
6               if left.rightY < right.rightY
7                  move left child into node spot
8               else
9                  move right child into node spot
10              end if
11           else
12              move left child into node spot
13           end if
14        else
15           if right subtree exists then
16              move right child into node spot
17           else
18              there is no child, remove node
19           end if
20        end if
21     else
22        mid = (minY + maxY) / 2
23        if A.rightY < mid then
24           node=left child, maxY=mid
25        else
26           node=left child, minY=mid
27        end if
28     end if
29  end while
```

Figure 2.5: Pseudo-code for a rectangle removal from PST.

Upon adding a rectangle, it is known that all active rectangles are intersecting over *X* coordinates, and *Y* coordinate based ranges are needed to be checked for an intersections of a

newly added rectangle with all of active rectangles. This operation is called enumeration and the pseudo-code is shown in Figure 2.4. Ideally this operation has $O(\log N+K)$ for a single rectangle. Enumeration can follow both right and left subtrees or just a right subtree.

Finally, a pseudo-code of a rectangle removal operation is presented in Figure 2.5. A rectangle is removed from PST upon hitting its right edge and optimally this operation has a cost of $O(\log N)$. Once the rectangle is found upon traversing the tree, it is removed and remaining nodes are into new positions.

2.3.3.2 Scan Priority Search tree algorithm

A fundamental solution introduces a plane sweep algorithm. A plane sweep algorithm, such as [46] can be described as a scan line sweeping across the problem space as in Figure 2.6. The direction of the sweep is arbitrary chosen. This dissertation uses a sweep from left to right on the X coordinate.

When a sweep line hits a left edge of a rectangle, the rectangle becomes active and is processed (e.g. tested for intersection) against existing active rectangles. A rectangle stays active until its right edge is hit. Therefore, vertical sides of rectangles represent events and there are $2N$ events for N rectangles in the dataset.

A fundamental $O(N \log N + K)$ implementation using [50, 47], hereafter is called SP (Scan Priority Search Tree) [57] – a variant of a Binary Search Tree (BST). We implemented a Y-coordinate based PST which ideally (when balanced) provides $O(\log N)$ tree operations which are insertion, search and removal. To emphasize, $O(\log N)$ is the optimal runtime complexity

that can be achieved, but only by adding more serial overhead (a sweep-line structure) and making sure the sweep-line structure is balanced.

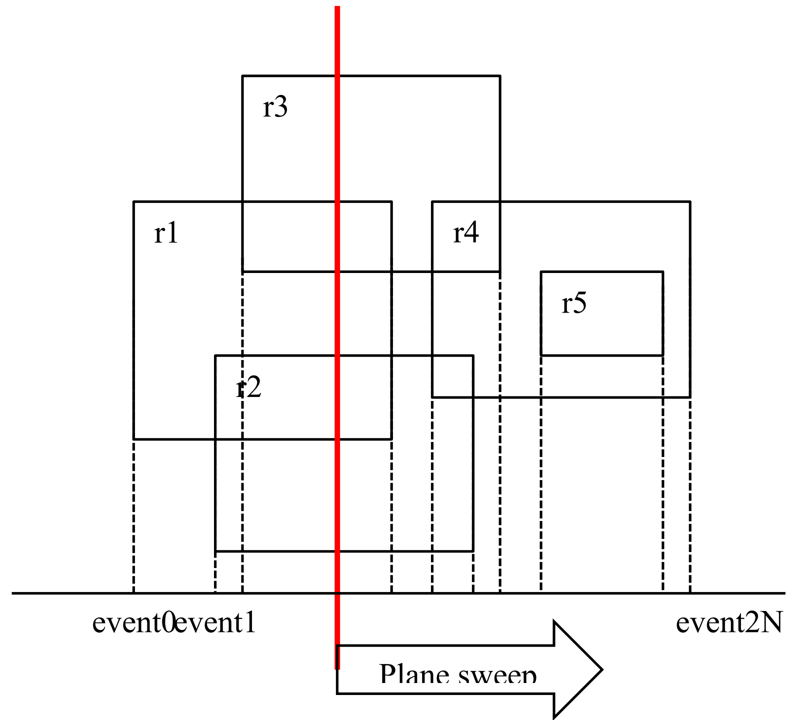

Figure 2.6: Scan Priority Search Tree (SP) algorithm applied to rectangles. From [30].

Our complete SP algorithm implementation consists of three parts: setup, event sort, and scan and has similarities to the one presented in [55] and shown in Figure 2.7. The setup step is where we extract left and right edges from rectangles. These edges represent events used while scanning. The SP solver constitutes the scan part. The original algorithm [50] requires both right and left edges of each rectangle (events) to be sorted in ascending order since having the rectangles presorted by *leftX* does not guarantee this. This makes the number of events to

be sorted to be *2N* as seen in Figure 2.6. The scan part is modified to match [50] where a rectangle is added into the pool on its left edge (*leftX* coordinate) and removed on its right edge (*rightX* coordinate).

To maintain *O(N log N + K)* running time we used the application-specific form of a generic PST from [47] as a data structure to organize rectangles in a pool. Enumeration of intersecting rectangles (checking for *Y* intersection) is done at every new insertion since only those rectangles that are currently present in the tree can intersect.

```
    Input: rectangle list rSet
    Output: rOutputSet
1   setup stage, extracting events (edges) to eventSet
2   sorting of events in eventSet
3   for ∀e ∈ eventSet do
4      if it is left edge then
5         insert rectangle into PST
6         compute intersections with active rectangles
6         add intersection rectangles to rOutputSet
7      else
8         remove rectangle into PST
9      end if
10  end for
```

Figure 2.7: Pseudo-code for SP solution.

2.3.4 Proposed algorithm

We call our proposed solution for the rectangle intersection problem Scan-List (SL). The direction of the scan is arbitrarily chosen. This dissertation uses sweep from left to right on the *X*-coordinate. Only left edges represented by *leftX* coordinates are considered while scanning. The scan step sweeps from left to right in the list, picking one rectangle (*R1*) at a time and checking for intersection with the ones to the right of this test rectangle. The scan continues as long as the right-edge coordinate of the test rectangle (*R1.rightX*) is greater than the left-edge coordinate of the newly picked rectangle (*R2.leftX*) as shown in Figure 2.8. This is the stopping condition since this violates (1). Scan is the only SL stage with no other overhead.

The worst case serial complexity for SL is $O(N^2)$. There are two factors that make SL competitive in a practical sense with optimal algorithms $O(N \log N)$ that use a sweep-line structure: first, the pathological case where all rectangles, or a large majority, can be on the sweep-line at the same time is not possible in ASIC design logic, and second, the lack of a sweep-line structure makes the constants in the secondary search time very low. We can neither assume that the rectangles are uniformly distributed. Remaining cases will have uneven slice densities over possible sweep directions. Since each layer is allowed only one specific routing direction [49] the sweep direction can be efficiently chosen to take advantage of such topology. Therefore, we pick the case which we consider the worst average case with $N^{1/2}$ rectangles in a slice. If we let $M = N^{1/2}$, we find that SL is $O(N*M + K)$ whereas optimal

sweep-line algorithms are *O(N log M + K).* As our experiments will show, the simplicity of the SL approach, and the reasonable assumption about rectangle distribution, cause SL to run as fast in parallel as sweep-line (SP) for reasonably large *N*.

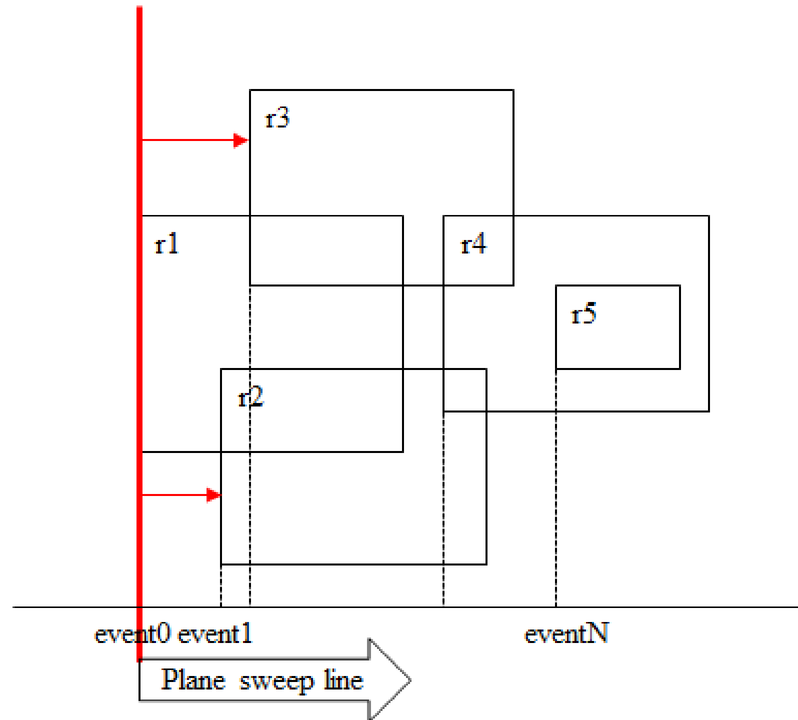

Figure 2.8: Scan-List algorithm visualized for rectangle R1.

We profiled a real-world 7 M rectangle dataset to verify our assumptions about the rectangle distribution pattern. The results are shown in Figure 2.9 with the mean distribution value being between 900 and 1000 rectangles in a random vertical slice. This is an average case that is lower than $N^{1/2}$ although some of the samples are in the upper range. The majority of the events are in 500-1000 range which is significantly lower than the worst case that we

assumed while justifying SL algorithm. This distribution pattern was also verified on other datasets as illustrated in the next sections.

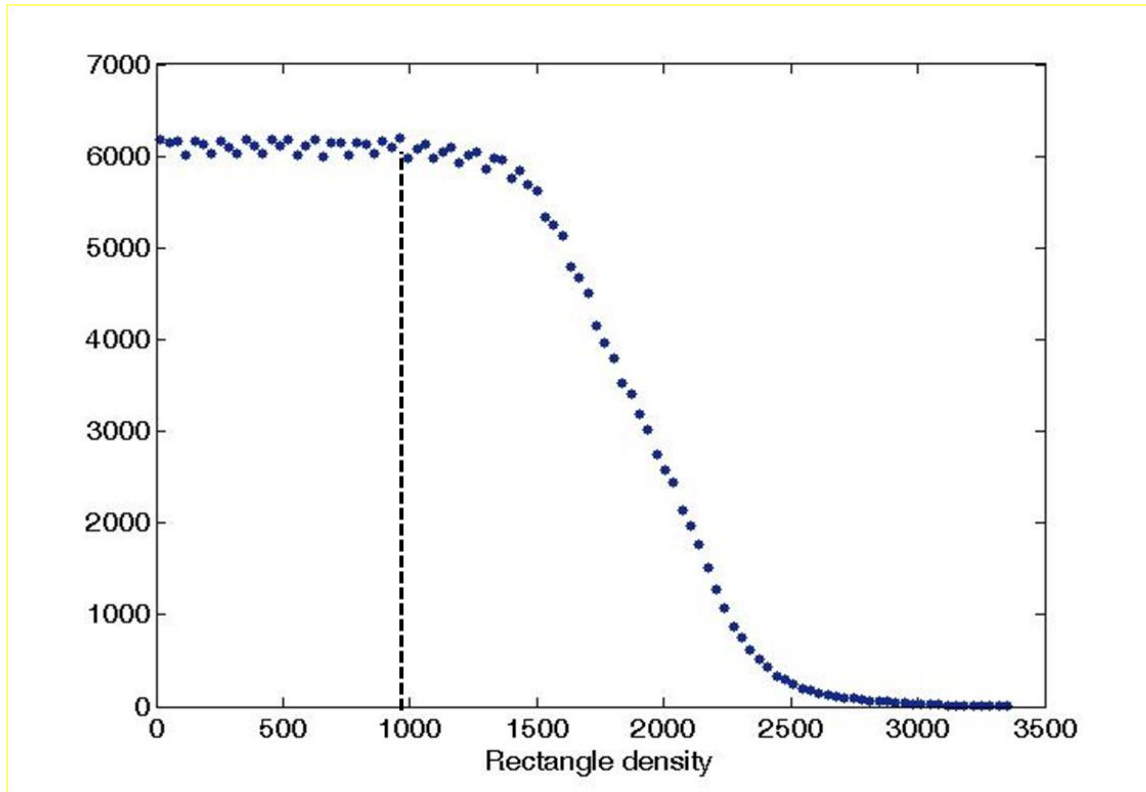

Figure 2.9: Rectangle density histogram for the 7 M test with mean. From [30].

SL pseudo-code is shown in Figure 2.10. Unlike $O(N \log M + K)$ solutions, SL does not require any complex sweep-line data structure such as segment or priority tree. Therefore, no expensive data structure operations or maintenance is performed. The SL algorithm operates with the original list of rectangles and achieves $O(N+K)$ space complexity with K representing number of intersections.

SL scan is basically a rectangle list iteration of picking up one rectangle at a time and checking for intersections with ones located further in the list. It is clear that the proposed SL algorithm should scale linearly because iterations through the rectangle list (the outer loop in Figure 2.10) are completely independent and can be done in parallel. The complexity of parallel SL, therefore, is assessed to be $O(M + K)$.

```
    Input: rectangle list rSet
    Output: rOutputSet
1   for ∀r1 ∈ rSet do
2     for ∀r2 ∈ rSet with index > R1 and R1.rightX >= R2.leftX do
3       if r1 and r2 intersect using (2.2) then
4         compute intersection rectangle;
5         add intersection rectangle to rOutputSet
6       end if
7     end for
8   end for
```

Figure 2.10: Pseudo-code for SL solution. From [30].

To complete complexity analysis, we assess I/O operations. Today shared memory platforms resemble PRAM described in [61]. Within the SL context, each thread reads a separate interval of rectangles from the main input and outputs a disjoint set of rectangles. The parallel SL does not produce duplicate intersections so no merge is required. Since the reads and the writes for each processor are completely disjoint from each other, this is a trivial case with the same number of I/O operations as in the serial version. There is no I/O contention except for the pure bandwidth limit of memory. Therefore, theoretically this algorithm scales

perfectly linearly and its performance would be ideally limited only by raw bandwidth available on the bus.

In conclusion, we can say that SL algorithm is in the embarrassingly or pleasingly parallel class of problems [60], problems that can be fully partitioned and require no communication between cores while being solved in parallel.

2.3.5 Efficiency analysis and serial run results

To understand potential scalability limits we profiled the serial implementations of the algorithms to observe run time breakdowns. We used an in-house synthesis design tool to map behavioral register transfer level (RTL) and generate netlists containing 1, 3, 7, 14 and 25 million rectangles. These netlists represented functional unit blocks of modern processor designs (courtesy of Intel Corporation) and reflected challenges currently faced in the semiconductor industry with most functional unit block sizes in the millions. Other types of design may use larger sizes and we address performance on these in section 2.4.2. The SP versus SL serial performance results are intuitive and reflect the running complexities that were analyzed in 2.3.3 and 2.3.4 and comparison is visualized in Figure 2.11. SL is as effective as SP for reasonably large problems due to additional overhead that is present in SP. For larger problems the comparative *serial* SL performance deteriorates significantly due to its *O(M)* versus *O(log M)* factor while scanning.

The detailed timings for SL and SP for tests are illustrated in Tables 2.1 and 2.2. Table 2.1 has breakdowns per SP effort. According to timing measurements both setup and sort are

significant. Therefore, efficient scaling of SP would require all three steps to be truly scalable. Table 2.2 has comparative details of SL experimental runs for given problem sizes with the run time of SL approaching $O(N * M + K)$.

Table 2.1:
SP running time breakdown, in seconds. From [30].

Problem	1 M	3 M	7 M	14 M	25 M
Setup	0.15	0.46	1.01	1.93	3.61
Sort	0.13	0.37	0.74	1.49	3.22
Scan	0.35	1.16	3.03	5.12	11.41
Total	0.63	1.98	4.79	8.53	18.24

Table 2.2:
SL running time, in seconds. From [30].

Problem	1 M	3 M	7 M	14 M	25 M
Scan	0.86	5.06	21.15	32.69	88.95
Total	0.86	5.06	21.15	32.69	88.95

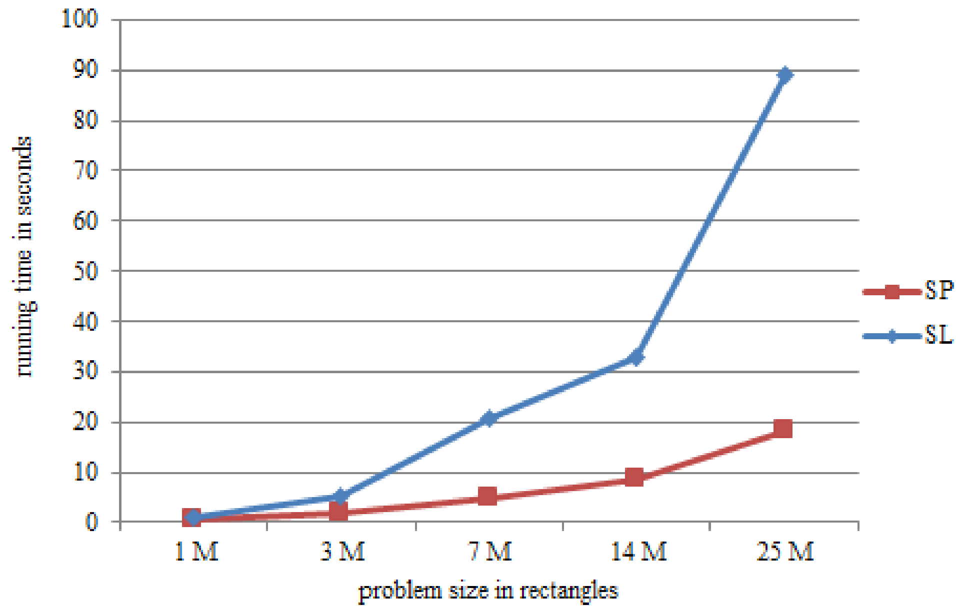

Figure 2.11: Serial run results comparison SP versus SL.

2.4 Parallel implementation and results

2.4.1 Implementation

The distribution of the rectangles is not uniform – there are lumps of concentrated logic with some die estate reserved for I/O and interconnect. This would introduce balancing issues across cores when sweeping. Therefore, we needed to include some heuristics into the solver for both algorithms to achieve optimal load balancing.

We applied the GNU parallel sorting algorithm [53] and optimized the comparison via inline functions for SP event sorting. To enable parallel execution of the algorithms we used

the OpenMP approach described in section II of [55] and [12]. The intersection reporting was done in a manner similar to thread local storage so that no inter-thread synchronization was necessitated. For the SP algorithm the implementation involved reprogramming most of the solver so that core scheduled data structures are self-contained to ensure as little synchronization between threads as possible. The parallelization effectively happens along the *X*-coordinate since the sweep traversed horizontal slices similar to what was done in [55] and shown in Figure 2.12.

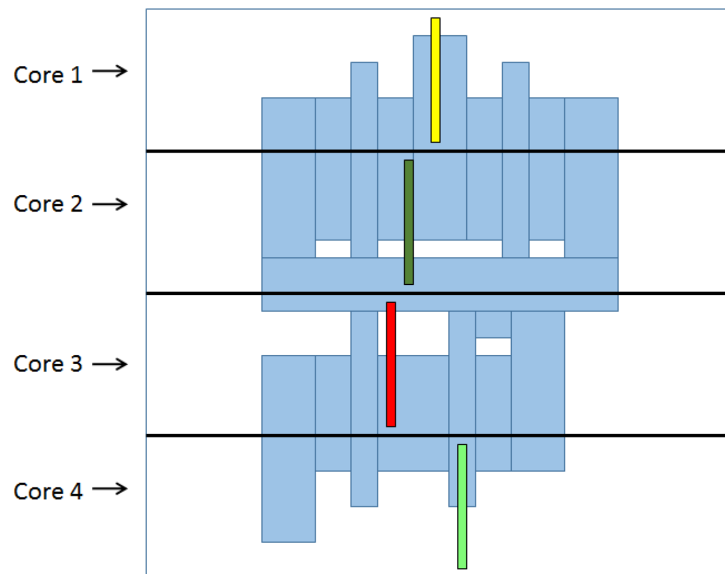

Figure 2.12: Load balancing issues with SP algorithm visualized.

We also changed the problem setup stage to address the balancing issue. Parallel sweeping required rectangles were assigned to each core (slice) similar to [55]. The naïve approach would be to slice the range of coordinates that rectangles occupy (min and max) over the

number of cores N. However, this scheme does not guarantee balanced partitioning since rectangles are not distributed in a uniformal manner as we see later on. Load balancing issues preventing all thread to finish the task synchronously are illustrated in Figure 2.13.

The conventional techniques to deal with the load balancing are static and dynamic scheduling. Dynamic scheduling can be done with help of work-stealing, i.e. when a thread that is much ahead and is light on the workload, steals work from other threads thereby trying to balance the execution. In the context of SP, Y-coordinate range would be adjusted to allow light-work threads been assigned more work. This technique is illustrated in Figure 2.13 and, notably, requires additional overhead.

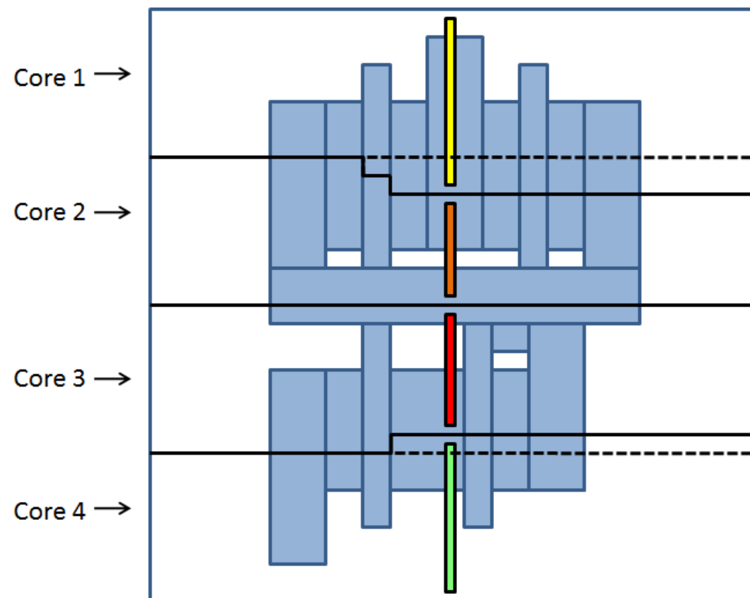

Figure 2.13: SP dynamic workload balancing with work-stealing.

Instead, we developed a static balancing approach that would have rectangles evenly distributed among the available threads so that no thread balancing (communication) occurs while sweeping in parallel. Each thread (slice) needed to be assigned a coordinate range. If a rectangle spanned over several slices, it was split appropriately. The pseudo-code for finding the slice coordinates is presented in Figure 2.14.

Input: *arrayMeanY, N*
Output: *sliceCoor*
1 divide arrayMeanY[numRectanles] by cores N to get sliceIncrement
2 **for** each core i **do**
3 find the slice coordinate sliceCoor[i] = arrayMeanY[sliceIncrement * i]
4 **end for**

Figure 2.14: Pseudo-code for finding slice coordinates for SP static balancing. From [30].

Table 2.3:
Maximum deviations for each scaling experiment. From [30].

Slices / threads	4	8	12	16
Deviation	0.05%	0.07%	0.19%	0.27%

We made the observation that a good heuristic in balancing would be to use rectangle means that were already computed in the setup stage. The *arrayMeanY* consisting of *Y* coordinate means for each rectangle allowed us to efficiently estimate load distribution. We used the slice increment *sliceIncrement* to calculate indexes. Then we indexed the *arrayMeanY* for each slice to get the slice coordinate *sliceCoor*. The cost of this balancing

scheme is minimal and is dwarfed by overall setup effort and allowed us to achieve close to linear speedup for the scan part in SP. Maximum deviations from ideal distributions of rectangles across all slices are presented in Table 2.3 for each scaling experiment and noted to be extremely low. Experimental results that are presented later show the scan part scaling nearly linearly confirming the efficiency of this load balancing scheme.

The event sort part of the solver scaled since the slice dedicated event lists were sorted in parallel for all threads at once. An additional post-processing step was required to merge intersections that happened due to split rectangles in neighboring slices. The merge (patching) effort was estimated in [55] to be *O(PN log PN)* time complexity for *N* rectangles and *P* slices (cores).

```
   Input: rectangle list rSet
   Output: rOutputSet
1  #do in parallel
2  for ∀r1 ∈ rSet do
3    for ∀r2 ∈ rSet with index > R1 and R1.rightX >= R2.leftX do
4      if r1 and r2 intersect using (2.2) then
5        compute intersection rectangle;
6        add intersection rectangle to rOutputSet
7      end if
8    end for
9  end for
```

Figure 2.15: Pseudo-code for parallel SL solution.

Contrary to what was required for SP, the effort to scale SL was minimal indeed with no additional overhead added. The parallelization of the outer loop was done using OpenMP [12]

as illustrated on line 1 in Figure 2.15. We natively cut the original iteration list into core dedicated sublists so that cache utilization is optimal. The slicing happens vertically in this case along *Y*-coordinate with no merge (patching) required since parallel SL does not cut any initial rectangles. To address the balancing issue, the OpenMP scheduler API schedule(type[,size]) size parameter can be tuned for an appropriate sublist sizing. Since this setting has to be large enough to support parallelism, this also eliminates any possible I/O conflicts in parallel SL by creating loop bounds. Thread specific storage vectors were used for storing intersecting rectangles to avoid thread synchronization.

2.4.2 Experimental setup

We used a four socket Intel® Xeon X7350 2.9 GHz [51] with a total of 16 cores, for the scalability experiments. The same hardware was used for the serial experiments. Each core a single threaded unit with 4 cores located on the same die (socket) providing an opportunity to validate SL on a shared memory platform.

To completely validate SL scalability and I/O analysis we needed larger cases that we were not able to generate using industrial physical design tools. We programmed a synthetic dataset generator RGener to produce needed datasets. The generator needed to produce datasets that would closely match real-world datasets since the algorithm scalability cannot be validated otherwise. Rectangle density distribution is the key (if not the only one) characteristic that exists.

Industrial EDA tools work hard trying to fit as much logic on a die as possible and this makes the problems much denser on some parts of a die that are allocated for custom designed logic with remaining parts reserved for routing. Figure 2.16 illustrates theoretical sweep line comparison on RGener and real-world (EDA tool) layouts with 10% intersection ratio (one intersection for ten rectangles). This means that a vertical slice and a structure that holds it would have more entities in real-world cases than in their synthetic counter partners if RGener is not tuned properly.

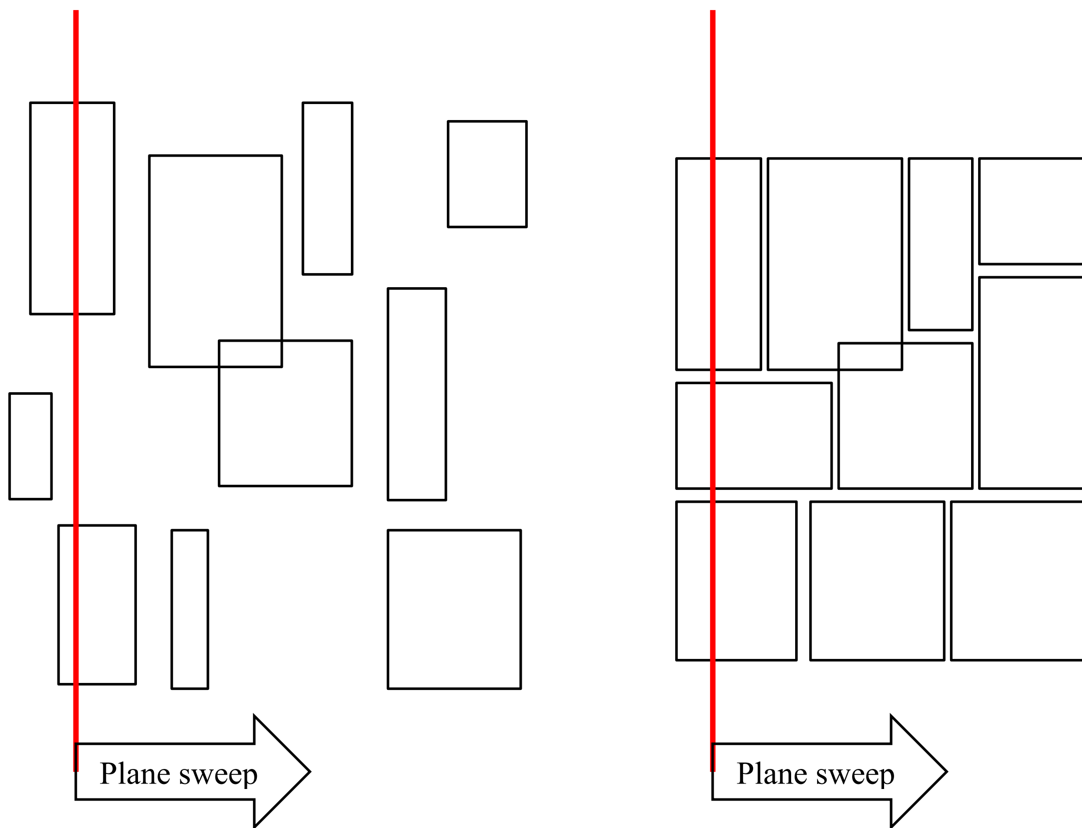

Figure 2.16: Differences of synthetic (left) and EDA (right) layouts.

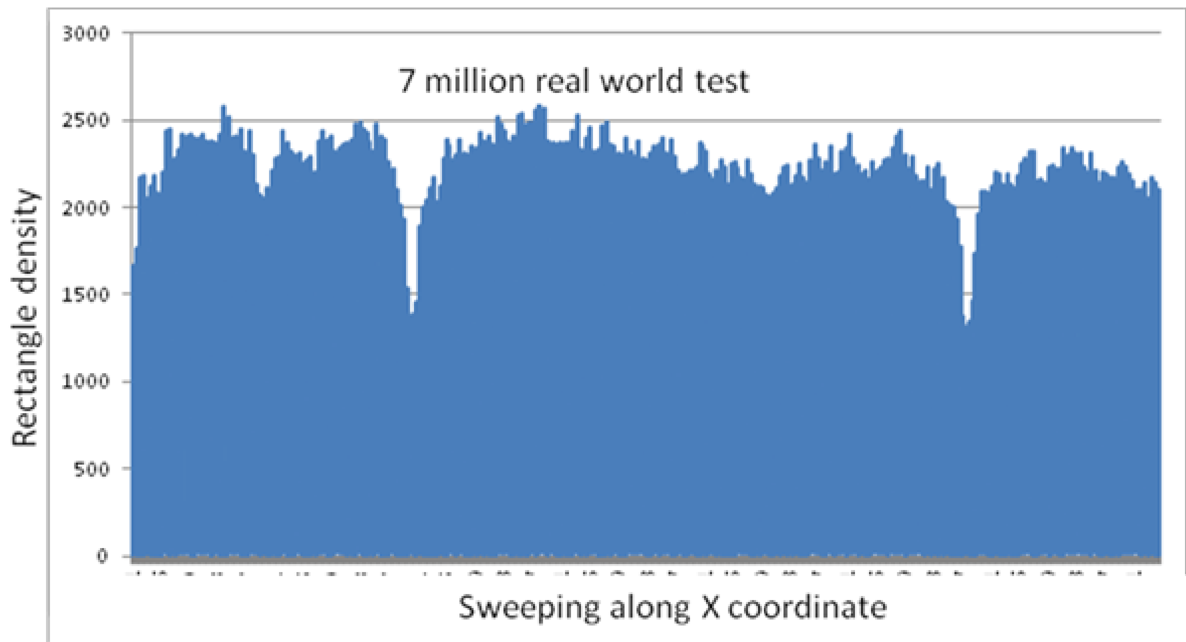

Figure 2.17: Rectangle density histogram for a fragment of the 7 M dataset.

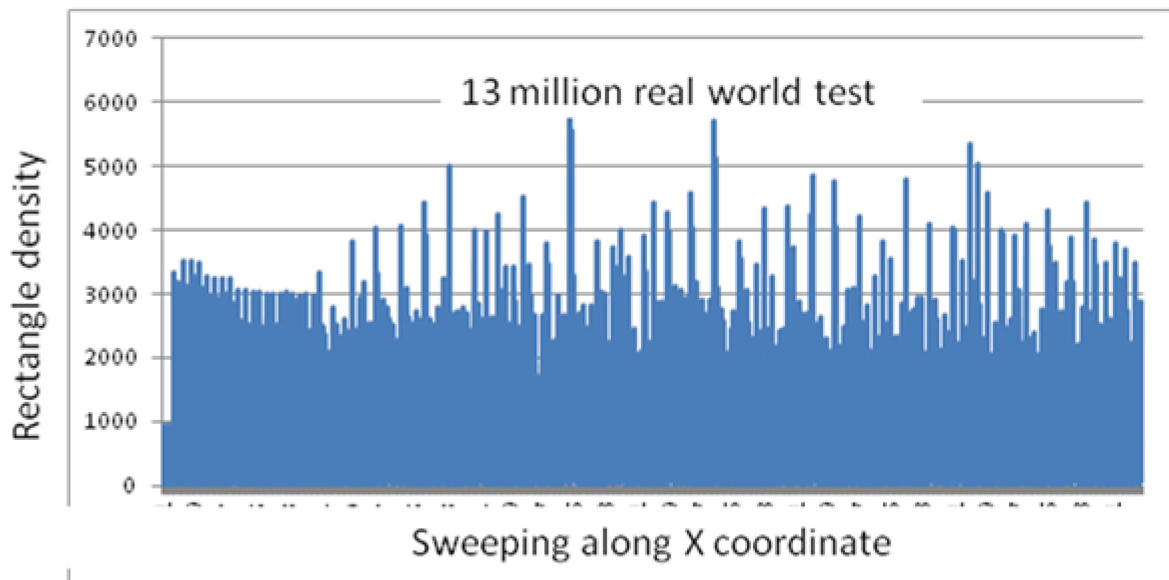

Figure 2.18: Rectangle density histogram for a fragment of the 13 M dataset.

Firstly, we profiled several real world datasets and the profiling results are shown in Figures 2.17 and 2.18. Repeatedly, the rectangle density distributions are less than the worst case assumptions and are in line with the results illustrated in Figure 2.9. In addition, rectangle densities are not uniformal as there are some high and low density areas. It is likely that the density spikes reflect clustered areas and in VLSI are associated with custom blocks such as caches or register arrays. The areas with lower rectangle densities reflect space reserved for chip interconnect. The datasets generated with the initial RGener settings met the average density that would test the complexity aspect of SL. However, the initial datasets had a relatively uniformal rectangle density distribution shown in Figure 2.19 which is different from real-world datasets and therefore not meeting the requirements to validate the algorithm efficiency.

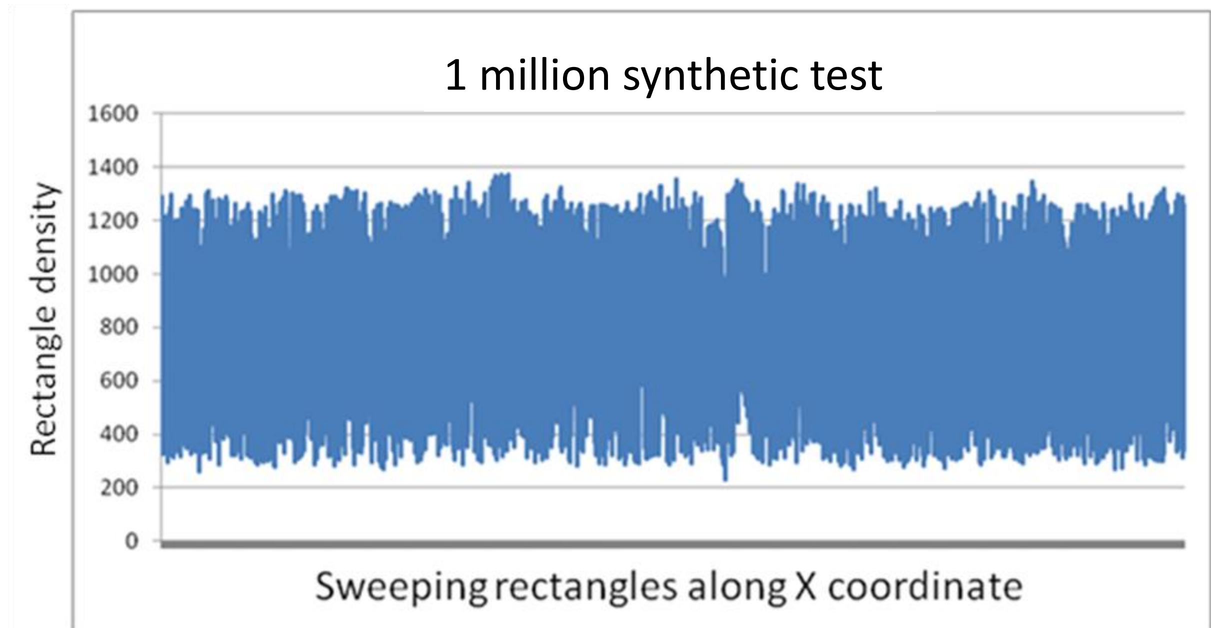

Figure 2.19: Rectangle density histogram for a fragment of the 1 M synthetic dataset generated with initial settings of the generator.

The distribution of rectangles is critical for the load balancing when processing a dataset in parallel because distribution irregularities cause load balancing issues and threads may not have equal work. This would cause suboptimal scalability and, therefore, suboptimal parallel performance. Therefore, we tuned our rectangle generator to closely match rectangle density distribution of real world tests with an example shown in Figure 2.20. The density distribution is not as uniformal as it used to be and there are some spikes that would stress the algorithm. We also generated 1 B rectangle tests that we used in scalability experiments.

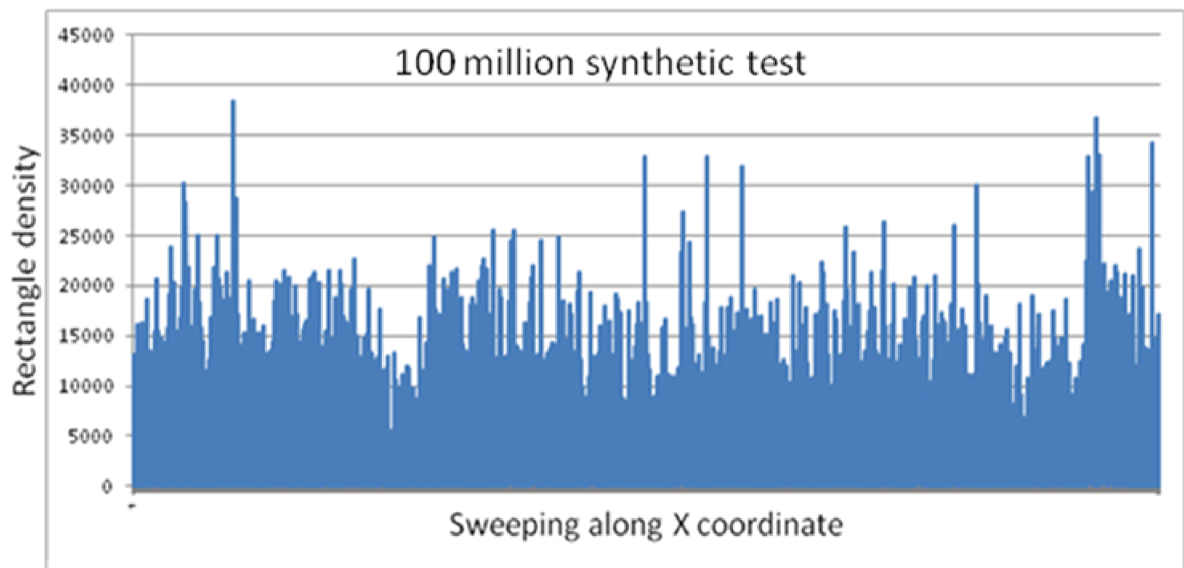

Figure 2.20: Rectangle density histogram for a fragment of the 100 M synthetic dataset generated with the final settings.

2.4.3 Results

We experimented with the 7 M test and achieved a speed-up for the SP scan part that goes along with results received in [12] which is around 6.5 on 8 cores. Table 2.4 has the complete parallel SP solution results that are far from ideal due to the presence of significant overhead as described in section 2.3.3.2.

Table 2.4:
Scaled SP performance for 7 M case, in seconds. From [30].

	Cores				Speedup on 16
	1	4	8	16	
Overhead	1.75	1.91	1.85	1.88	N/A
Scan	3.03	0.85	0.48	0.22	13.91
Total	4.79	2.75	2.33	2.10	2.28

This goes along with results that were achieved in [13] with approximately 2.5 speedup. The setup stage effort does not increase in higher core count runs and this confirms our previous estimates on the balancing scheme complexity. A higher number of narrower slices will be needed to have SP working on higher core count machines. This makes initial problems larger with additional rectangles due to splitting and increased merging as in [13]. We therefore note that it is mathematically impossible to achieve linear speed-ups for the SP

algorithm on high core count machines. Parallel SL results are shown on Table 2.5. A nearly linear speedup was achieved for larger problems.

To better illustrate SL scalability we used data from Tables 2.4, 2.5 and plotted rectangles per second on Figure 2.21. SL has a linear performance and outpaces SP at around 10 cores for the 7 M rectangle test.

TABLE 2.5:
Scaled SL performance, in seconds. From [30].

Rectangles	Cores				Speedup on 16
	1	4	8	16	
1 M	0.86	0.31	0.16	0.10	8.46
3 M	5.06	1.43	0.81	0.45	11.12
7 M	21.15	5.57	2.74	1.37	15.44
14 M	32.69	8.31	4.18	2.10	15.59
25 M	88.95	22.59	11.33	5.64	15.78
100 M	1716.95	431.99	217.17	108.67	15.80
1 B	62018.40	15503.20	7804.54	3893.85	15.93

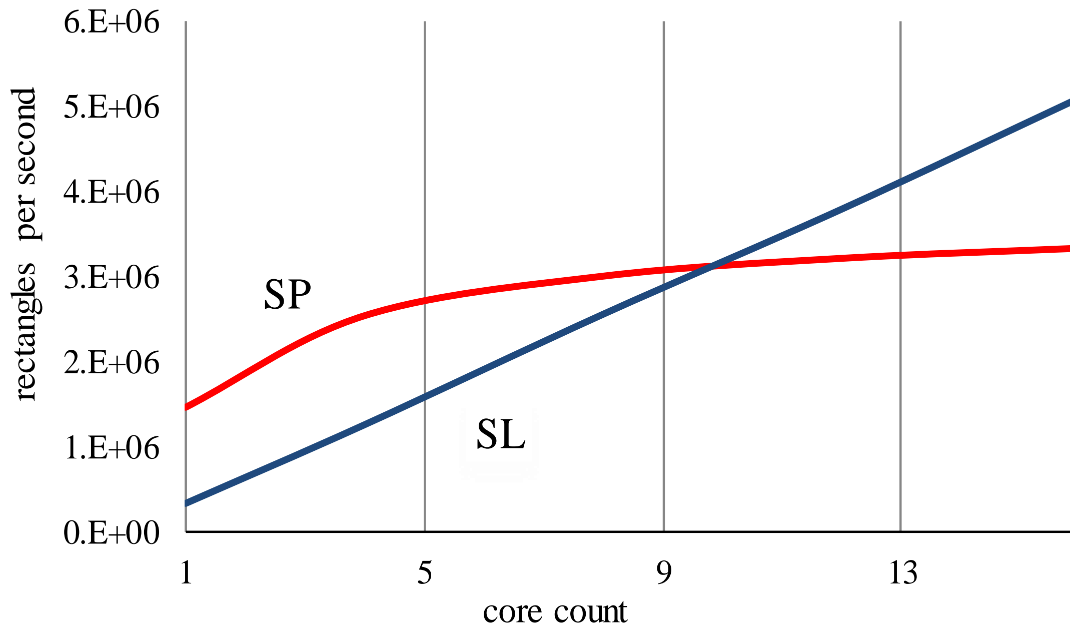

Figure 2.21: Scaled SL vs scaled SP for the 7 M rectangles test. From [30].

We were also interested to see how well scaled SL performs against scaled SP for larger problems. We used data from Tables 2.4, 2.5 and additional simulation data points to create a graph of a scaled SL versus SP crossover line shown on Figure 2.22. A test with around 30 M rectangles would be the largest size that SL could match SP on 16 core machine [9]. It would require 39 cores for SL to match SP for the 100 M test. Provided SP has serial overhead it is expected that the slope coefficient for the crossover line would be less than one. We extrapolated data points in order to verify the assumption about the slope coefficient and visually it is calculated to be close to 0.8. If we provided additional data points on higher core count machines, the crossover line would resemble a hockey stick with the slope coefficient

being constantly reduced. These estimates provide us with assurance that scaled SL will outperform SP on future large problems with higher core machines available.

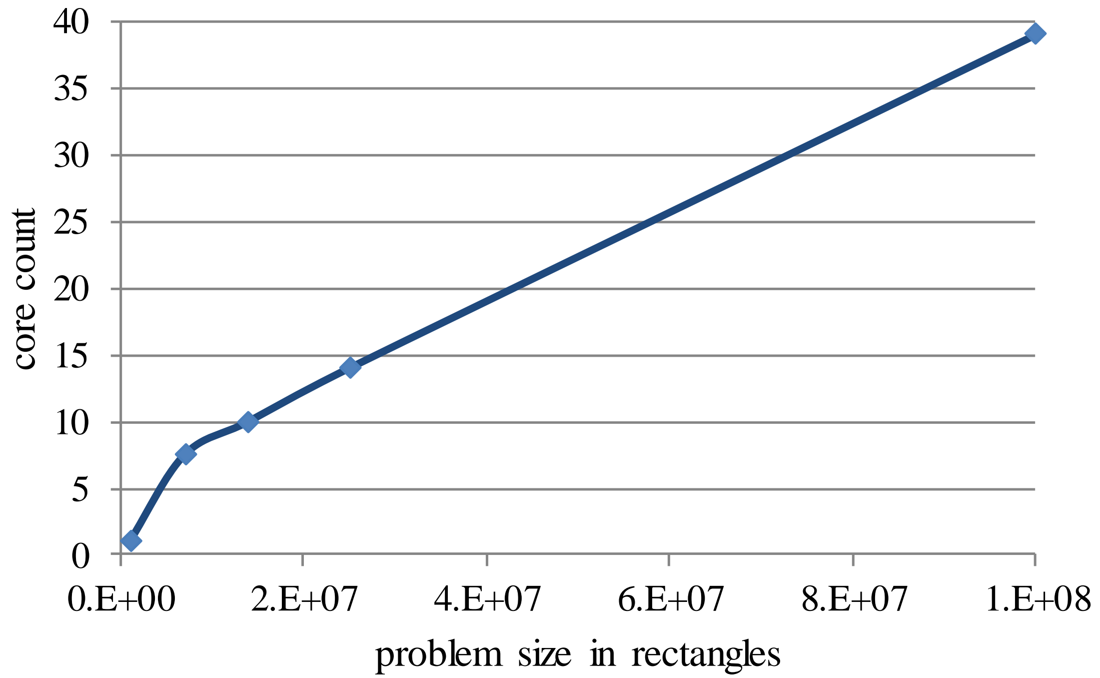

Figure 2.22: Scaled SL vs scaled SP crossover trendline. From [30].

Chapter 3

Optimizing Parallel Label Propagation based Community Detection on the Intel® Xeon Phi™ Architecture

3.1 Motivation

Complex systems such as social, biological and information networks, characterized by millions to billions of sub-entities and relationships between them are best represented by graphs. A distinguishing feature of such complex systems is the self-organization into dense clusters called communities. Detecting such communities in massive graphs is critical to the understanding of such complex systems. However community detection is non-trivial and is expensive due to a vast number of computations needed to fully realize the underlying relationships. While a number of near-exact and heuristic algorithms is available for community detection, parallelizing such algorithms to fully leverage the advantages of parallel hardware is still a challenging problem. Most presently available approaches attempt to optimize run-time complexities by scaling original serial community detection algorithms. Graph algorithms in general and existing community detection algorithms in particular are

known for not being commensurate with linear scalability on parallel systems. Therefore, there is a need for a combination of high-performance software and a hardware platform that would support efficient parallel graph processing with respect to community detection. The Intel® Xeon Phi™ Label Propagation algorithm (PLPA) variant of community detection algorithm based on label propagation (LP) is presented in this chapter. PLPA was tuned for the Intel® Xeon Phi™ platform a novel architecture that provides 50+ physical cores with simultaneous multithreading. Phi™ architecture advantages and limitations for PLPA and massive graph processing in general are outlined in this section of the dissertation. Test results of running PLPA on large real-world networks while achieving near linear speedups and improving the quality of the detected communities are illustrated as well in this chapter. Further possibilities of processing massive networks that cannot fully fit in a Phi™ memory are analyzed and hence a modified PLPA (PLPA-M) extends the initially proposed community detection algorithm on Phi platform.

3.2 Introduction

Sparse graphs [10] is a particularly effective way to model various networked systems such as social [2], biological [4, 5], informational [3] and other networks [6, 7]. Graphs that represent such systems have their distinctive characteristics. One common feature is that the graph structure comprises of clusters, or communities, with a relatively high count of edges connecting nodes inside communities and a lower count of inter community edges that connect the communities themselves [8]. Another distinctive feature of sparse graphs is that

node degrees follow power-law distribution and having long tail pattern as in Figure 3.1. The most of nodes in a networked system have small degree counts while much smaller number of nodes have high degree scores. The latter are defined as hub nodes shown in Figure 3.2. Hub nodes constitute a base of each cluster and generally have much greater influence on their neighbors than otherwise. Hub nodes and nodes that they are strongly connected to comprise a cluster or a community. A key general feature of a community is that its entities are very likely to share similar properties. In biological networks, collection of individual entities into functional modules facilitates understanding of molecule roles [16]. In social networks, hobbies and political, sport and purchasing preferences are typical community properties, just to site a few. These properties together with hub nodes provide valuable information about structures of networks and communities that constitute them. The role of a fast and reliable way to perform community detection in comprehending the structure of massive graphs cannot be understated. The same is reflected in the attention received by community detection algorithms by researchers across a wide range of disciplines. Considering that graphs is a conventional way to represent communities, one may think that community detection and graph partitioning are similar problems. The goal of graph partitioning is to divide a graph into approximately equal size groups of nodes.

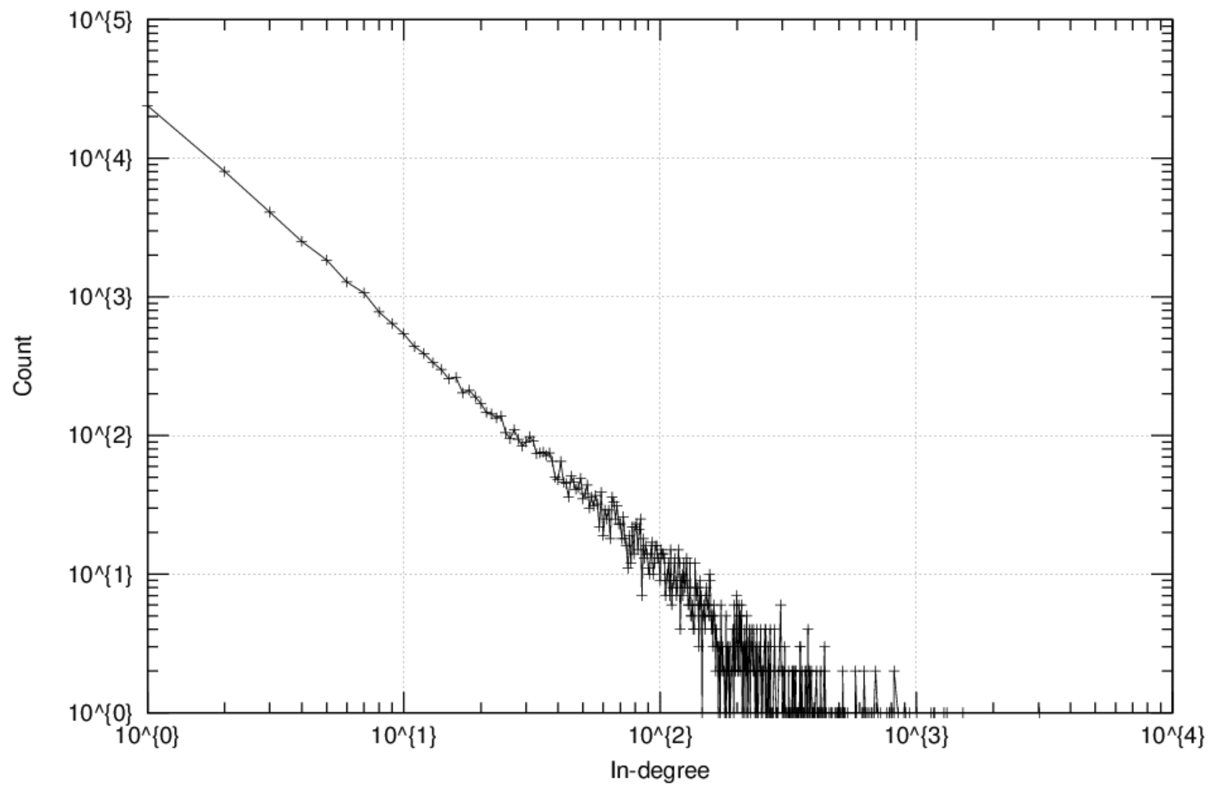

Figure 3.1: Example of sparse graph degree distribution (base-10 log-scale). Who-trusts-whom (75K, 508K) network of Epionions.com from SNAP.

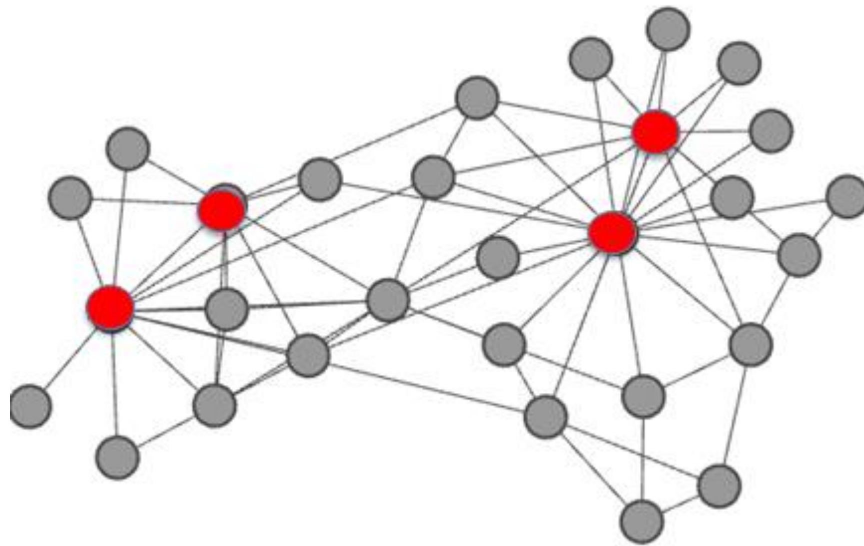

Figure 3.2: Example of a small network with “hub” nodes.

On the contrary, community detection is an unsupervised method that identifies groups of nodes with similarities. Graph clustering is a natural way to perform this task and has been studied in computer science for decades. The number of groups is unknown at the start and is identified in the process. Figure 3.3 shows a flow of a typical clustering algorithm. Upon parsing of a graph a basic topology is defined. At this point nodes and edges of the graph are known as well as node degrees. The second graph in Figure 3.3 illustrates nodes sized according to their degrees. During the next step a graph clustering algorithm is applied. A clustering algorithm can be an iterative one when it is applied to the network many times until the network converges to a stable state or until some stopping condition. Once a clustering algorithm converges, each of nodes in a graph belongs to a community. There is a remaining

step of measuring goodness of detected communities and there are several measures that are covered later in this dissertation.

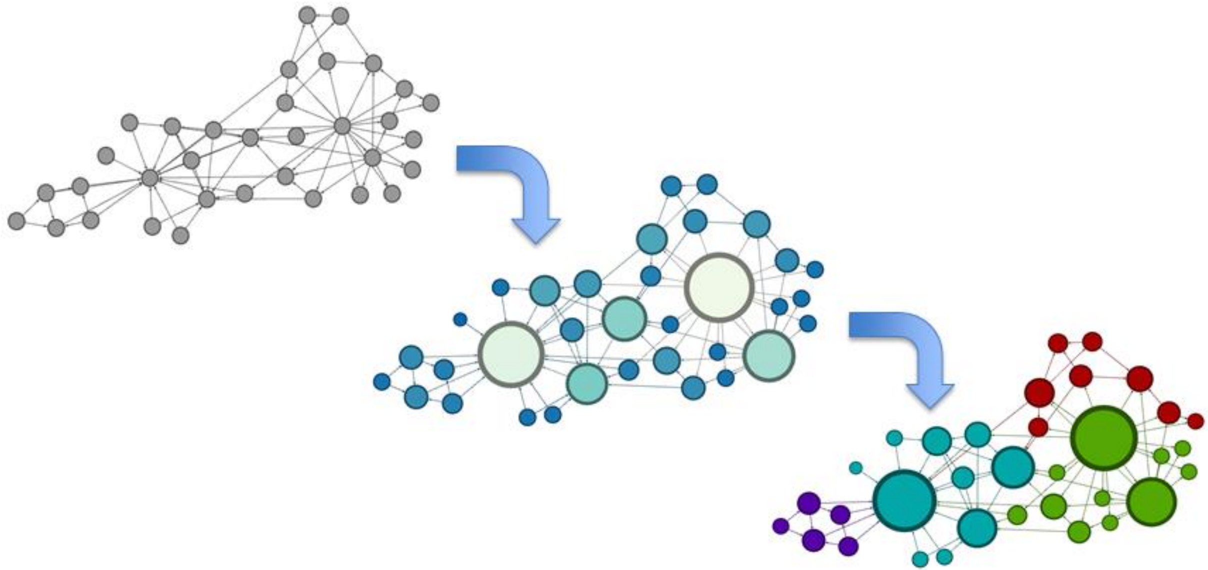

Figure 3.3: A community detection process visualized.

The most popular serial algorithms along with their complexities and clustering quality measures are well summarized in the survey by Fortunato [10]. Presently, a typical *online social network* (OSN) can include millions to billions of nodes and edges [9]. *Modularity* has been a popular clustering quality measure. However, graph clustering for *modularity optimization* is known to be NP-complete [31], and this makes community detection in OSN via graph clustering to be a non-trivial task.

Several efficient heuristics have been presented that significantly reduce the complexity of community detection [10]. The most popular community detection algorithms fall into major

categories of *divisive* or top-down algorithms [8], *agglomerative* or bottom-up algorithms [20, 24], *modularity maximization* algorithms [26] and a category of various alternative approaches such as *label propagation algorithm* (LPA) [17]. *Modularity* based approaches include spectral optimization [22], simulated annealing [21], greedy routing [19] and a compression-based method [23], just to cite a few. The two key performance metrics of all community detection algorithms are complexity and quality of final clustering. There are several popular community clustering measurements [10], and modularity arguably has been the most popular one. Modularity defines the goodness of a community by comparing the fraction of edges within partitioned clusters to a "null model" that is characterized by random edge distribution model placement. Modularity function peaks when clustering achieves ideal identification of communities as seen in [24]. A modularity score of 0.3 or above usually means a strong community structure. Typical values are in 0.3 to 0.7 range. However, given its optimization complexity, modularity is conventionally used as a measuring tool and not as a driver. As for the algorithm complexities, they range from being close to linear complexity of $O(N)$ for [17] to $O(N^2)$ or worse in the vast majority of other methods. Therefore, it is logical to affirm that only those algorithms with near $O(N)$ complexities can be practically considered to process massive graphs.

Until recently, Intel® Xeon and AMD Opteron™ modifications have been the conventional shared-memory platform choices for parallel implementations. These systems scale up to 32 physical cores and have been mainstream of parallel exploration for more than a decade. However, the parallelism of computing systems has advanced to another level with the first generation *Many Integrated Core* (MIC) product *Intel® Xeon Phi*™ (Phi) [37].

Various Phi configurations provide 50+ physical cores with simultaneous multithreading (SMT) allowing to span 200+ hardware threads. A Phi can be packaged with Intel® Parallel Studio (Studio) that facilitates creating reliable parallel code [41].

3.3 Related work

There has been significant amount of the previous and related work in community detection algorithms and a good summary of these achievements is illustrated in [34]. Around half if not

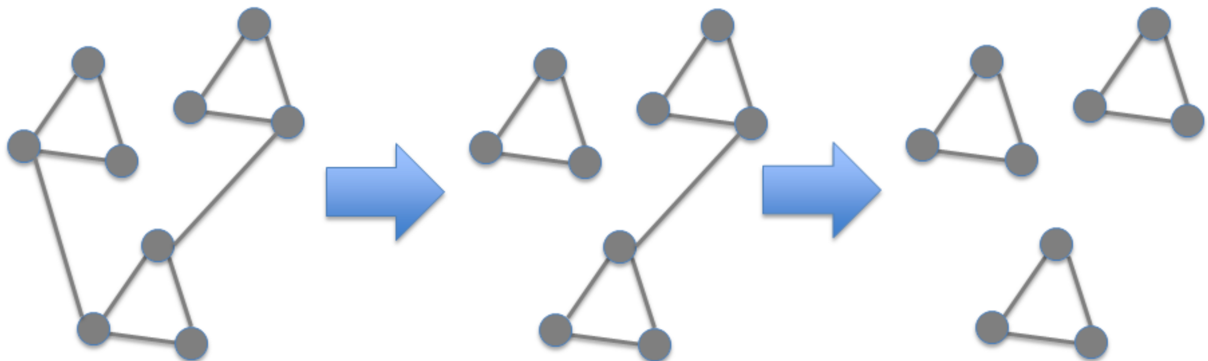

Figure 3.4: Girvan-Newman edge betweenness based community detection algorithm visualized.

most of the existing clustering methods have high runtime complexities. For example a well-known edge betweenness [24] has a cost of $O(M^2N)$. It is based on an iterative search for the

edge that has the highest flow in the current network topology as shown in Figure 3.4. The search itself has an approximate cost of $O(M^2)$. Once the edge is located, it is removed from the network, modularity for a new topology of this network is calculated and a new search is performed. This process continues until a stopping condition that can be a specific number of removed edges or a significant drop below the achieved maximum of a modularity score. The beauty of this method is that deterministically partitions the network and it does this in very logical way. Serial implementations of this algorithm suffer from inherited poor performance due to a high complexity cost. Notably, it takes close to 300 seconds to iterate over the network with five thousand nodes and fifteen thousand edges. This timing results are extremely poor in comparison with the methods presented later in this dissertation.

Approximation, incremental and parallel modifications have been popular ways to improve algorithm run-times. Incremental methods are best suited for dynamic networks with continuously updated topologies, whereas we focus on static systems in this dissertation. In order to have efficient parallelization, an initial serial algorithm must be virtually free of serial overhead as Khlopotine *et al.* illustrated in [30]. Otherwise, incremental improvements to a serial algorithm most likely will contribute to performance degradation while running in a parallel manner. Parallelization of inherently complex serial algorithms such as [24, 27, 28, 29] have limited benefits thereby moving research focus to clustering algorithms that are as simple and as fast in a serial mode as possible. Arguably, Louvain [26] and LPA [17], both being non-deterministic algorithms with close to linear time complexities, have been getting the most of the attention recently. Both Louvain algorithm and LPA have very little to no overhead and are based on *iterative local update,* i.e. when vertices are moved to neighboring

communities in order to maximize modularity. Louvain method is at $O(M)$ for a single iteration and is visualized in Figure 3.5. Louvain method searches for the best option (a greedy method), namely the best community, where to move a node. Once a node is moved, it becomes a part of the community to where it moved which means it is essentially merged with a neighboring node. For a node v, the number of options is bounded by the degree of v. Modularity is acting as the cost function. Upon the first pass, the topology of the network is being recalculated and merged nodes are organized into so-called "super" nodes. The iterating process continues until there is no incremental improvement for the network modularity. In addition to its multilevel approach, the incremental modularity calculation has a high constant and adds to overall computation effort and becomes serial overhead.

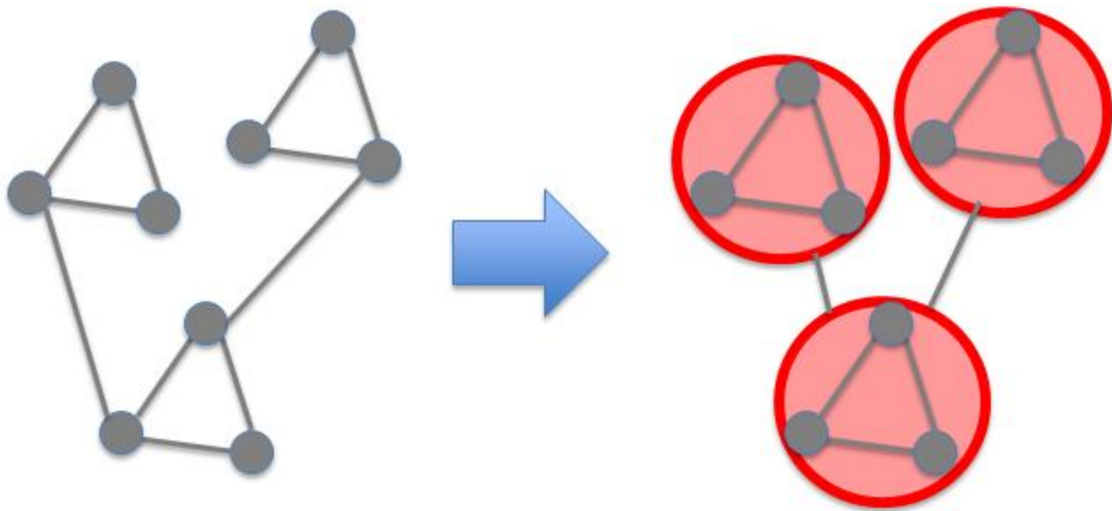

Figure 3.5: Louvain community detection algorithm visualized.

There are other components that prevent linear scaling of parallel Louvain as illustrated in [1, 32, 33]. To the contrary, LPA has an "epidemic nature" where a choice of the node

reassignment is dictated solely by the heaviest, most frequent label in the neighborhood of the node. Modularity does not affect the flow and is conventionally calculated upon the algorithm completion. Therefore, LPA is practically an overhead-free algorithm that shall scale linearly. Notably, there have been several parallel modifications of LPA algorithm that achieve sub-linear speedups on conventional multicore [32, 34, 36] and distributed memory systems [35] that we analyze later in this chapter.

As we have already discussed, there are several metrics for goodness of a community [34]. We have adopted the modularity approach since it has been the most popular metrics and also there have been developed a number of modularity maximization methods where modularity is included as a cost function.

3.4 Label Propagation Algorithm

3.4.1 Definitions

We define a graph $G = (V, E)$, where V is a set of vertices (or, interchangeably, nodes) of size n and E is a set of edges of size m that connect the vertices. V and E define respectively the order and the size of G. Vertices u and v are neighbors if there is an edge e such that $(u,v) \in E$ connecting them. $NB(v)$ constitutes the neighborhood of v. The number of neighbors $D(v) = |NB(v)|$ of a vertex v constitutes the degree of v. Maximum degree in the network is defined as $D_{max} = Max(D(v)),\ \forall v \in V$. Node labeling is defined as $L(V)$ such that $\forall v \in V$, $L(v)$represents a community label of each node. The modularity function Q can be defined as

(1), where M is an adjacency matrix. The function δ yields one for u and v being in the same

$$Q = \frac{1}{2m} \sum_{uv} \left(M_{uv} - \frac{D(u)D(v)}{2m} \right) \delta(C_u, C_v) \qquad (1)$$

community and is zero otherwise. Graphs can be weighted and unweighted, directed and undirected, static and dynamic or any combination of these. Notably, some fraction of graphs are undirected, unweighted and static as shown in Figure 3.6. For simplicity this dissertation deals with unweighted static graphs [10] with a goal of identifying non overlapping communities. This work can be easily extended to weighted graphs.

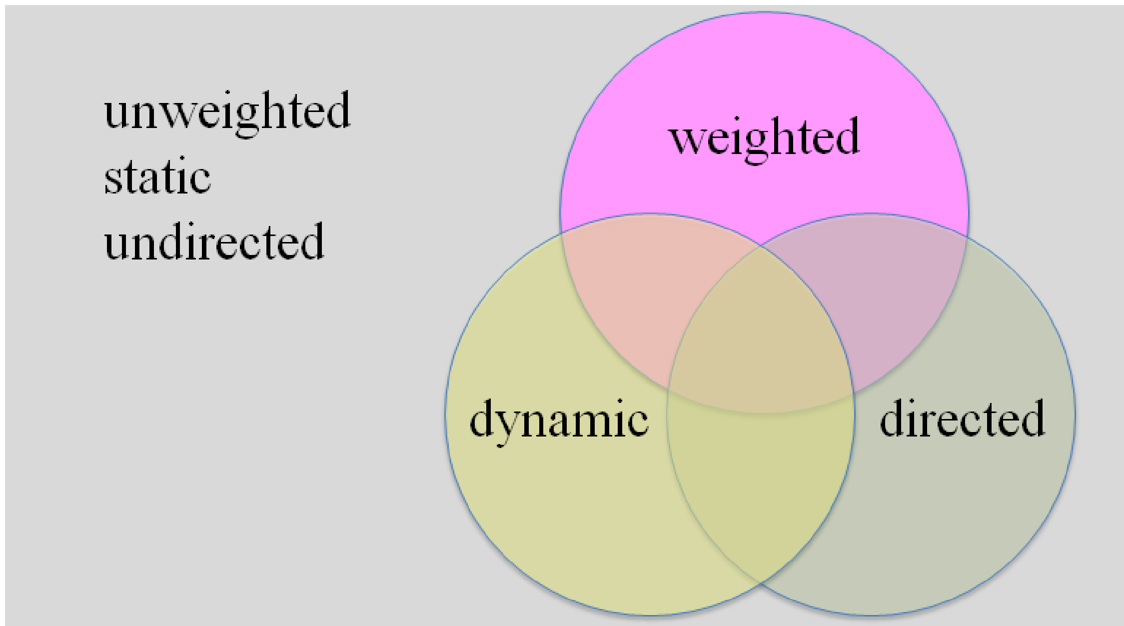

Figure 3.6: Common types of graphs.

3.4.2 Label Propagation Algorithm

LPA, as originally introduced by Raghavan *et al.* [17], produces graph clustering by assigning or labeling v to a specific community defined by the label. Initially each node v is assigned a unique label that could be an index of v in V with values in $\{1, \ldots, n\}$. Therefore, the initial number of clusters in the system is equal to the number of nodes n. In each iteration i a node v changes its label $L(v)$ based on the most frequent label in $NB(v)$. Therefore, label updates happen in an "epidemic" manner and the algorithm converges once the number of clusters N stabilizes, i.e. *upd* being the number of nodes that changed assignments is less than a non-negative threshold value Δupd. The output of LPA is a list of final assignments $L(V)$. Conventionally,

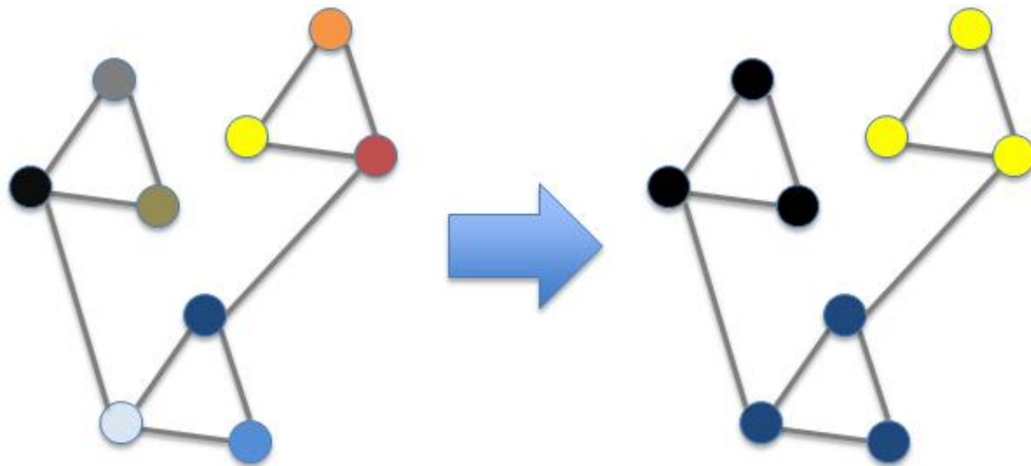

Figure 3.7: An example of LPA processing a nine node network.

the node traversal order and label ties in *NB(v)* are picked randomly. An iteration count depends on the randomization strategy, a node ordering and topology characteristics of the network and there is no published analysis on how correlated these factors are. Our empirical evidence reported that on average each iteration reduces the number of active nodes by a binary factor. Therefore, on average it takes 20 iterations for LPA to converge on a network with one million nodes. A visualization example of LPA on a simple six node graph is shown in Figure 3.7. In this example each node is assigned a unique color (instead of an index) in the beginning for better visualization. There might many coloring possibilities upon the convergence of LPA on this network with one of the options presented in Figure 3.6.

In another example in Figure 3.8 we illustrate a six node network that has a hub node that is colored green. Theoretically this network can be finally colored in any color. Practically there is more than 80% of probability that the final coloring will be green since hub nodes have higher influences on the final coloring pattern than other nodes in the network. The influence would directly correspond to network properties such as hobbies or purchasing patterns. In reality, nodes are interconnected to thousands of other nodes and the six node graph would be a part of a larger network as illustrated in Figure 3.8. and, therefore, other nodes properties would be involved in the labeling process.

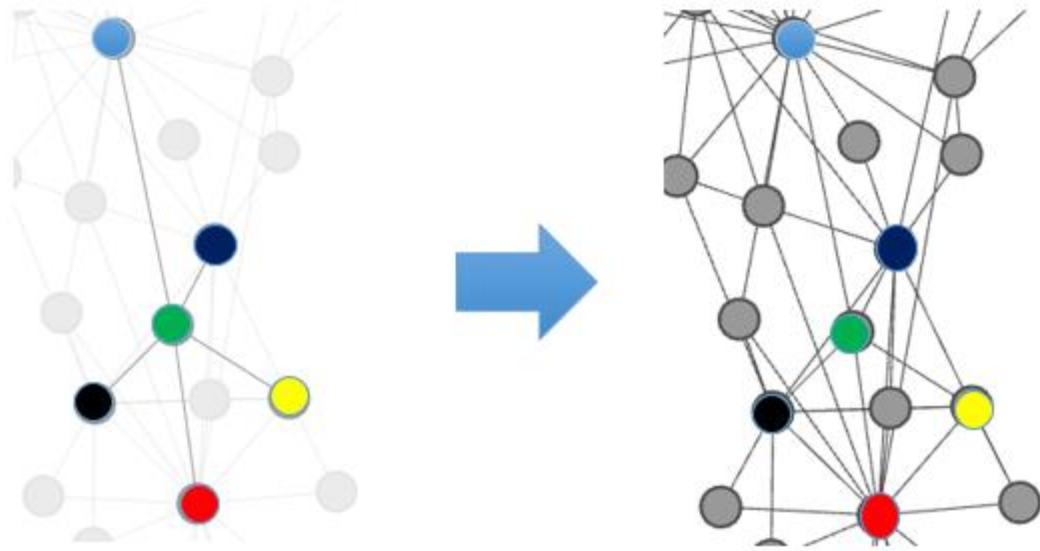

Figure 3.8: A low level node connectivity illustration example. The figure was generated using Gephi network analysis tool.

A pseudocode for LPA is shown in Figure 3.9. The update process can be either synchronous or asynchronous. The synchronous approach uses *L(V)* assignments from the previous iteration and may cause oscillation as described in [17] thereby increasing LPA convergence time. In addition, the quality of detected communities may suffer because of the use of dated node information. To the contrary, the asynchronous variation always looks up the most recent label. Therefore, it is important that line 5 in Figure 3.9 uses the current *L(v)* value. Given a sparse nature of complex systems such as OSNs, each LPA iteration complexity is close to *O(m)*.

There have been several improvements to the initial LPA. Staudt *et. al* [32] proposed reducing a problem size by maintaining an active list of nodes *A(V)* and limiting nodes that

are processed to only those that are in $A(V)$ and not isolated. Initially, this optimization puts all nodes into $A(V)$. Then, a node v is removed from $A(V)$ if $L(v)$ was not changed in the last iteration. If $L(v)$ or $L(u)$ such as $\forall u \in NB(v)$ was updated in the last iteration, v is put back in $A(V)$. This optimization significantly reduces the problem size since $|A(V)| << n$ after few iterations, and this was empirically proven in [32]. Liu *et al.* [14] proposed a semi-synchronous

```
     Input:  G(V,E), Δupd > 0
     Output:  L(V)
1    Initialize: ∀v ∈ V: L(v) ← v, up ← Δupd+1
2    while upd > Δupd do
3       upd ← 0
4       for ∀v ∈ V do
5          L_temp = Max_L( ∑(u ∈ NB(v))
6          if (L(v) ≠ L_temp then
7             L(v) ← L_temp
8             upd = upd + 1
9          end if
10      end for
11   end while
```

Figure 3.9: LPA pseudocode. From [62].

LPA modification with a goal of eliminating label oscillations in bipartite networks. This LPA flavor requires an additional graph coloring phase that is executed after the initialization and before any node can be processed. The coloring phase places nodes in colors C_j so that for $\forall v, \forall u \in C_j$ there is no edge $e(v,u) \in G$. The number of colors j is in a range of $\{1,...,c\}$ and is bounded by D_{max}. Depending on the coloring scheme and the seed value, the value of c can be smaller than D_{max}. To reflect this addition, a coloring effort and an extra loop that traverses

all colors is added to LPA. Notably, the complexity of each LPA iteration remains *O(m)* since the only difference is the order in which nodes (edges) are processed. Various serial and parallel approaches for graph coloring are made available [25] and greedy coloring arguably has been the most popular one with a computational cost of $O(|V|+|E|)$.

The mentioned optimizations maintain quality of detected communities according to modularity analysis presented in [14, 32]. Therefore, these optimizations shall be considered in the parallel version of LPA and we illustrate the pseudocode in Figure 3.10.

```
     Input:   G(V,E), Δupd > 0
     Output:  L(V)
1    Initialize: ∀v ∈ V: L(v) ← v, upd ← Δupd+1, A(v) ← 1
2    Coloring phase: ∀v, ∀u ∈ Ci , e(v,u) ∉G
3    while upd > Δupd do
4       upd ← 0
5       for ∀Ci ,  i = {1, … , c} do
6          for ∀v ∈ Ci and D(v) > 0 and A(v)==1 do
7             L_temp = Max_L( ∑(u ∈ NB(v))
8             if L(v) ≠ L_temp then
9                L(v) ← L_temp
10               upd = upd + 1
11               ∀u ∈ NB(v), A(u) ← 1
12            else
13               A(v) ← 0
14            end if
15         end for
16      end for
17   end while
```

Fig 3.10: Optimized LPA pseudocode. From [62].

3.5 Experimental setup

3.5.1 Intel® Xeon Phi™ platform

Major advantages and limitations of a Phi platform are illustrated in [11, 42]. A platform consists of a Xeon series host processor and a Phi coprocessor. Xeon and Phi are interconnected via x16 PCIe 2.0 protocol that allows up to 8 GB / sec for data transfers. The platform can be ordered with various configurations with one of the options being a platform with multiple Phi cards. A high level Phi platform diagram is shown in Figure 3.11. A Phi co-processor can be used as separate computing node or an additional processor in a heterogeneous type of system.

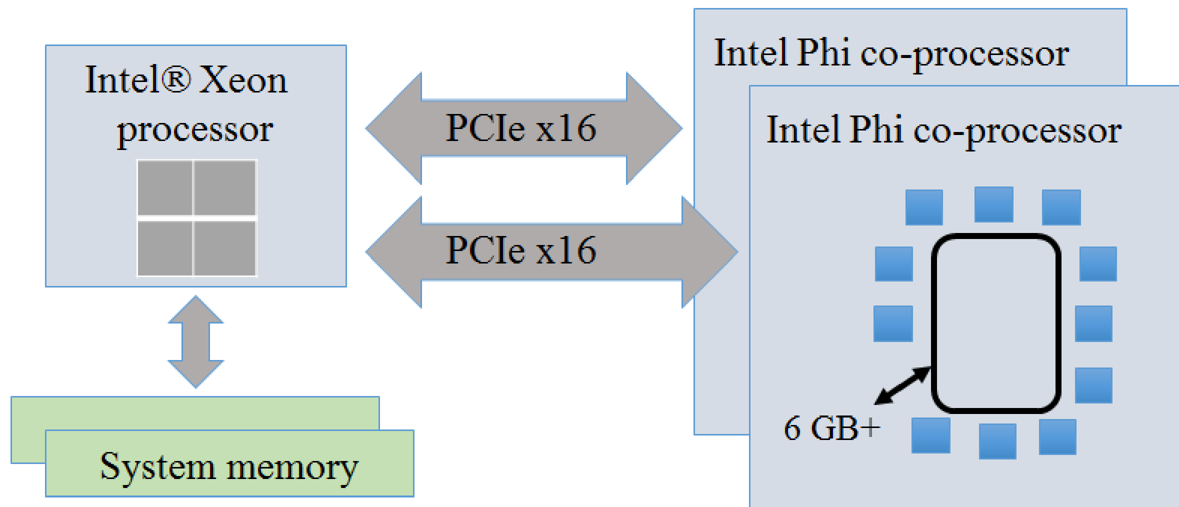

Figure 3.11: Intel® Xeon Phi™ high level platform architecture.

There are several programming models applicable to Phi. The offload model fits best the type of computing we are aiming for since it provides an opportunity for heterogeneous computing and utilizing a divide-and-conquer approach for work partitioning. There are several programming APIs available on a Phi co-processor. These APIs include OpenMP, MPI and IntelClk. Each Phi card can be used as a node similarly to nodes in a distributed systems.

Table 3.1:
Intel® Xeon Phi™ platform specifications. From [62].

	Xeon E5-2609 v2	Phi 3120A
Cores	4	57
Threads (SMT)	1 / core	4 / core
Frequency	2.5 GHz	1.1 GHz
Cache line	512 bits	512 bits
L1 cache	32 KB / core	32 KB / core
L2 cache	256 KB / core	512 KB / core
L3 cache	10 MB	N/A
Execution	Out-of-order	In-order
Power	80 W	300 W
SIMD width	256 bits	512 bits
Memory	GDDR3	GDDR5
Bandwidth (max)	42.6 GB / sec	240 GB / sec

Our Phi starter kit [38] brings 57 physical cores with one core been reserved for system processes. Each core is an updated version of the original Pentium p54C processor with hardware tuned to run parallel workloads. Two major updates are a four-way SMT and a novel 512 bit *single instruction multiple data* (SIMD) pipeline. With SMT turned on, our Phi system can span up to 224 hardware threads. Theoretically, SMT and the extended SIMD shall provide four and two-fold speedups respectfully, but this requires appropriate code modifications [42]. The kit is equipped with 6 GB of random-access memory (RAM) which is the minimum RAM configuration, and it can be expanded on higher end Phi systems. The host side is a conventional Xeon E5-2609 v2 socket. Notably, a Phi coprocessor is not equipped with hard disk memory which becomes a limitation of its processing power. We summarize our platform specifications in Table 3.1. With a price point that is close to $1600, arguably, Phi is a highly cost-effective solution for desktop parallel computing.

3.5.2 Networks and methods

We choose the largest available SNAP datasets [43] with their details presented in Table 3.2. The selected networks are real-world and represent several types of systems with various clustering coefficients, degree distributions and densities.

We make an observation that the first LPA iteration executes a similar workload across multiple runs on a single network independently of a node ordering. To the contrary, counts of subsequent LPA iterations and their timing measurements can vary significantly due to a

non-deterministic nature of LPA. We, therefore, measure speedup of the first LPA iteration to have precise scalability measurements. We run each experiment ten times and use average values in all of our reporting. We validate our LPA implementation by comparing total run times on Xeon in a serial mode with the results obtained in [32, 34].

Table 3.2:
Graphs used in the experiments. From [62].

	Type	Nodes	Edges
YouTube (YT)	ground-truth	1,134,890	2,987,624
Pokec (PO)	Social	1,632,803	30,622,564
LiveJournal (LJ)	Social	4,847,571	68,993,773
Orkut (OR)	ground-truth	3,072,441	117,185,083
Friendster (FR)	ground-truth	65,608,366	1,806,067,135

3.6 Label Propagation on Phi

3.6.1 Initial implementation on Intel® Xeon series

There have been several attempts to parallelize LPA [32, 34, 36, 35] all resulting in sublinear scaling. Therefore, we first implement a scheme to achieve linear scaling on the conventional Xeon architecture before we move to programming for Phi.

There are many factors that may prevent an overhead-free application from scaling linearly. Memory locking is one of them. Research work published in [34] claimed that the semi-synchronous approach minimizes memory locks that take place otherwise in the asynchronous approach while updating shared structures such as *A(V)* and *L(V)*. We argue against the same since for $\forall v, \forall u \in C_j$ it is not guaranteed that their neighbor lists are exclusive that is in general $NB(v) \cap NB(u) \neq 0$. Therefore, memory locks will occur when corresponding *A(V)* list locations are updated from different threads simultaneously. In addition, locks can occur on *L(V)* list since for $\forall v, \forall u \in C_j$, *L(v)* and *L(u)* updates can be performed from different threads, and it is not guaranteed that *L(v)* and *L(u)* are not located in the same cache line *cline*. Finally, having C_j colors with *j* in {*1,...,c*} will require LPA to execute a fork and join pattern causing additional synchronization that has a negative effect on scaling.

The probability of a lock *P(l)* happening on *L(V)* list is assessed in equation (3.1) with $|T|$ being a total number of available threads, n_u representing the number of nodes with updated labels, n_c is the number of cached *L(V)* elements, *b* is a size of the *cache* block and *n* is the number of nodes in the dataset. A lock would happen if multiple threads concurrently try to modify *L(V)* elements that reside in the same cache block. $(|T|-1) / |T|$ is the probability that the unassigned thread processes a node. N_u can be equal or, conventionally, is significantly less than *n*.

$$P(l) = \frac{|T|-1}{|T|} \cdot \frac{n_u}{n} \cdot \frac{n_c}{(n^* b)} \qquad (3.1)$$

N_c can be further assessed as *c_size* * *c_slice / v_size* where *c_size* is the size of the cache, *v_size* is the total memory allocated for single elements of list *L(V)* and *c_slice* is the proportion of the cache allocated for a sublist of *L(V)*. Given there are three more cached sublists of *D(V)*, *A(V)* and *NB(V)*, *c_slice* is assessed to 1 / 4 in the worst case scenario where each node has a single neighbor. Otherwise, for an average case with an average *D(V)* significantly larger than 1, *NB(V)* list would dominate, thereby reducing *c_slice* coefficient. It follows that large *n* would dominate in *P(l)* and reduce the probability of a lock.

Table 3.3:
Xeon run results. Total serial LPA time and the first iteration scaling measurements (in seconds). From [62].

	YT	PO	LJ	OR	FR
Total LPA	2.8	14.53	26.53	84.43	1,251.5
1 core	0.49	1.35	3.92	12.15	203.22
4 cores	0.21	0.37	1.07	3.27	51.32
Speedup	2.33	3.46	3.66	3.72	3.97

Hence, we choose the asynchronous method as a base for PLPA. To enable parallel execution we used C++ with OpenMP [12] approach similar to what was presented in [32]. We used static OpenMP scheduling and lower grain sizes for better load balancing, which we confirmed by code profiling with the Studio. Our implementation carefully minimizes use of shared structures by buffering most of the data with use of local structures. The empirical

results of are reported in Table 3.3. A linear speedup was achieved with the larger problems. Thread synchronization and load balancing constraints prevented optimal scaling on the smaller graphs. We also present total LPA execution measurements to better understand motivation for LPA optimizations and to confirm that our results are in line with the results obtained in [32, 34].

3.6.2 Optimizing LPA on Phi

There are several programming models for Phi, and these are presented in [42]. In general, Phi shall be utilized for highly parallel parts of code while leaving all overhead calculations to a faster Xeon series host processor. Choosing among the programming models, the explicit offload model is the best fit for LPA due to Phi memory capacity limitations. For example, the OR network in a user-follower input file format can use close to 2 GB of memory along, not counting allocated LPA data structures. This reduces the size of a network that can be processed on Phi. The explicit offload Phi programming model allows to selectively offload needed data structures to Phi while keeping the rest on a host. It also delegates execution to Phi while the host can execute another workload. This model works for applications that must be operating using shared memory paradigm. It is also possible that an application is developed in a way to run both on Xeon and Phi simultaneously. This is similar to running a parallel application on a distributed type of architecture.

There are several code modifications to be made in order for PLPA to run on Phi using the explicit offload model. All data structures need to be allocated on Phi before the first PLPA

iteration. The memory stays allocated for all following PLPA iterations through the last

```
     Input:   G(V,E), Δupd > 0
     Output:  L(V)
 1   Initialize: ∀v ∈ V: L(v) ← v, upd ← Δupd+1, A(v) ← 1
 2   Phi allocation: ∀v ∈ G, L(v), D(v), NB(v), A(v)
 3   while upd > Δupd do
 4     upd ← 0
 6     for ∀v ∈ G and D(v) > 0 and A(v)==1 do
 7       L_temp = Max_L( ∑(u ∈ NB(v))
 8       if L(v) ≠ L_temp then
 9         L(v) ← L_temp
10         upd = upd + 1
11         ∀u ∈ NB(v), A(u) ← 1
12       Else
13         A(v) ← 0
14       end if
15     end for
16   end while
17   send L(V) to Xeon for analysis
```

Figure 3.12: PLPA pseudocode. From [62].

iteration. Upon the end of community detection, Phi will send final *L(V)* assignments to Xeon for future analysis such as modularity computation. Given that each *L(v)* is allocated four bytes of a memory, a significant share of 6 GB memory allocated for other PLPA structures and given an 8 GB peak value for Phi interconnection bandwidth, a typical data transfer takes less than a tenth of a second. Once a data transfer is finished all data structures on Phi are deallocated. PLPA pseudocode is shown in Figure 3.12.

In addition to the structural changes, there is a need to ensure proper code vectorization [42] to fully benefit from the extended SIMD feature on Phi. Unlike a scalar-oriented code, a

vectorized code significantly reduces overheads by using bulk processing. Our adjustments include allocating 512 bit aligned data structures and recoding all *for* loops for implicit vectorization. We also placed *#pragma SIMD*, *#pragma ivdep* and *#pragma vector aligned* in front of each *for* loop to ensure compiler is aware of the optimizations.

Larger networks, e.g. FR, which due to its size cannot fit into the Phi memory, shall be processed by other methods such as PLPA-M presented later in this work. The scalability results for the remaining networks are presented in Figure 3.13 with PLPA reportedly scaling near linearly on larger datasets up to 32 cores. We profiled PLPA with the Studio and attribute performance degradation when utilizing more than 32 cores to the Phi on-die interconnect (ring) saturation. This reasoning is in line with the manufacturer (Intel®) information on performance hits for bandwidth-bounded applications with PLPA being one of them. We note that Phi measurement results vary by an average factor of eight when compared to the results that we achieved on Xeon and reported in Table 3.3. There are several architecture factors that contribute to the performance degradation. The Phi core operates at 2.3 times lower frequency and fetches an instruction every other cycle when compared to the Xeon core. The other major Phi disadvantage is the in-order execution pipeline that limits execution efficiency of complex memory access workloads [42] with LPA being one of them. A typical large network has $D(v)$ ranging to as high as hundreds to thousands neighbors $u_i \in NB(v)$ and most of u_i reside in distant memory locations. These placements prevent PLPA from achieving an optimal unit-stride access and spatial locality of reference which leads to extensive cache thrashing and a memory latency [11, 42]. However, this does not cause much of performance degradation on a Xeon architecture due to the out-of-order execution feature that hides the

issues seen on Phi. Finally, there are other recent hardware improvements of a translation lookaside buffer (TLB) and branch prediction schemes [42] making a Xeon core more attractive when compared to a Pentium Pro based Phi core. Notably, the next Phi generation, code name KNL for Knight Landing, is based on out-of-order execution cores [39]. In addition, the ring architecture that is present in the first generation of Phi will be obsolete once KNL is launched.

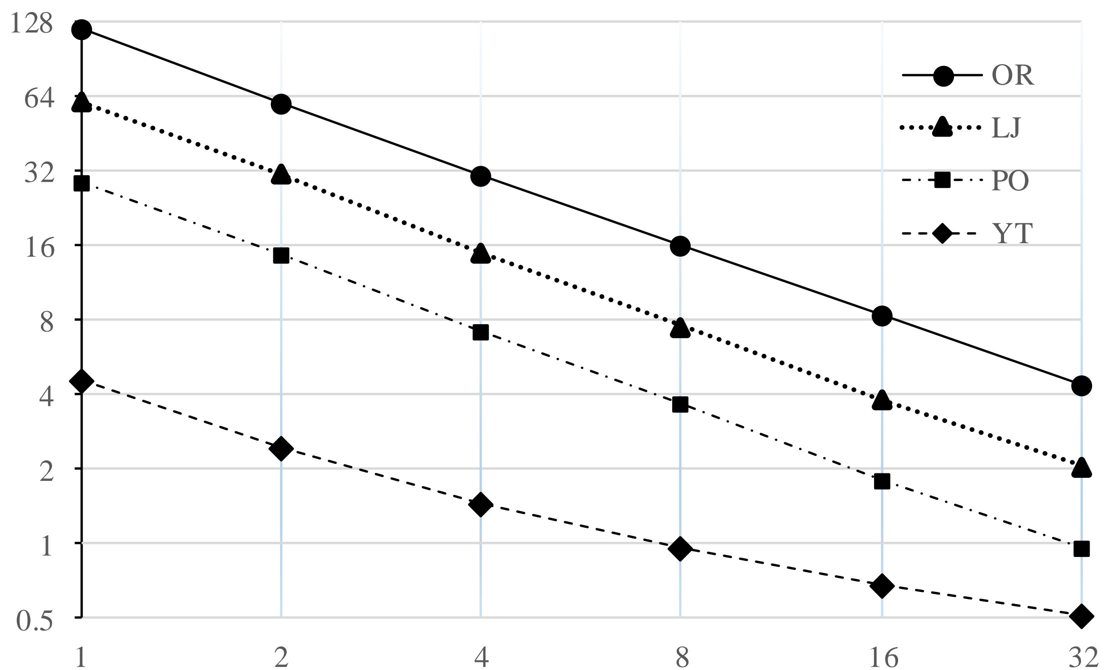

Figure 3.13: Strong scaling of PLPA (x-axis, core count, base-2 log-scale) running time in seconds (y-axis, base-2 log-scale). From [62].

KNL architecture is based on a mesh interconnect approach as shown in Figure 3.14. PLPA would definitely benefit from this architecture since the ring traffic saturation would

not be an issue any longer. These two KNL features shall significantly reduce the performance difference of PLPA on Phi versus Xeon while likely improving scalability on runs with beyond that 32 cores. Notably, PLPA could be run as is on KNL without any modifications.

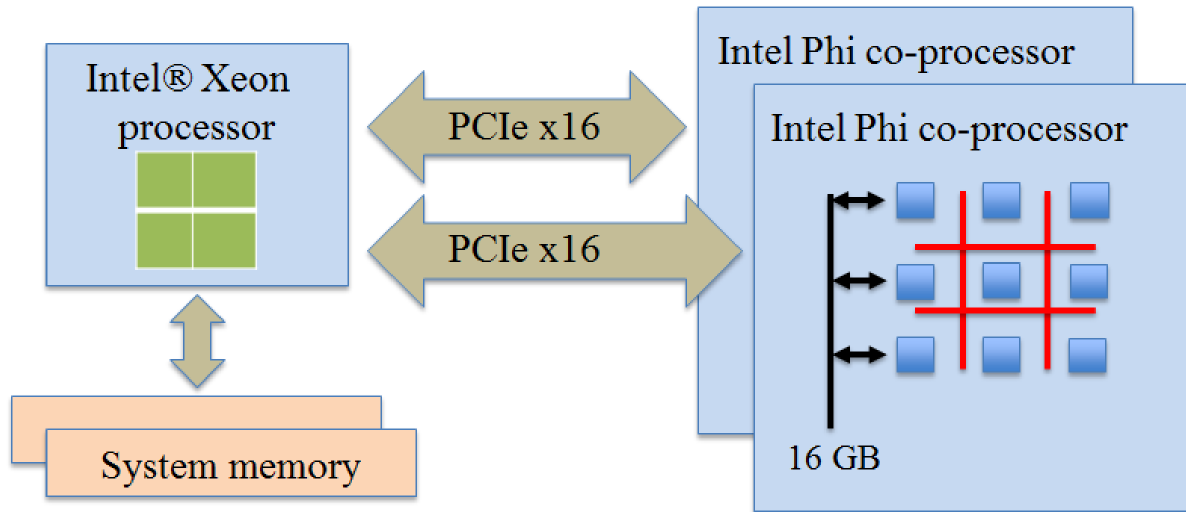

Figure 3.14: Intel® Xeon Phi™ high level platform architecture, the second generation.

However, there are two ways to optimize the memory latency that takes place with PLPA on our Phi platform. One way would be to reorder nodes in a way that guarantees an assignment of consecutive memory locations to a maximal number of neighbors. This requires extensive updates to the PLPA parser, and it is beyond the scope of this dissertation. An alternative way to mitigate the memory latency is to enable SMT. It can be done by using a value *compact* for the *KMP_AFFINITY* platform setting. According to the present research [42], three to four threads per core is an optimal combination for applications with performances suffering due to memory latencies.

Table 3.4:
PLPA results with SMT turned on modified networks. The first PLPA iteration measurements (in seconds). From [62].

	YT-M	PO-M	LJ-M	OR-M
1 thread	6.85	47.31	95.97	232.81
1 core, 4 threads	2.45	14.3	30.72	70.42
224 threads	0.98	0.81	1.34	2.27
Maximum speedup	6.99	58.4	71.62	102.56

However, we are constrained by the granularity of our test datasets, e.g. OR network being the largest one to fit in Phi memory is not large enough to efficiently scale on a maximum of 224 Phi threads. We enlarge the networks by guaranteeing for $\forall u \in NB(v)$ there is a bi-directional edge, i.e. $(u,v) \in E$ and $(v,u) \in E$. This modification is similar to the method presented in [32]. The modification increases m by a factor of two at maximum. We rename the modified networks by adding "M" to initial titles and report timing measurements with SMT turned on in Table 3.4. We achieved 103 speedup on OR-M that utilized only 20% of the Phi memory.

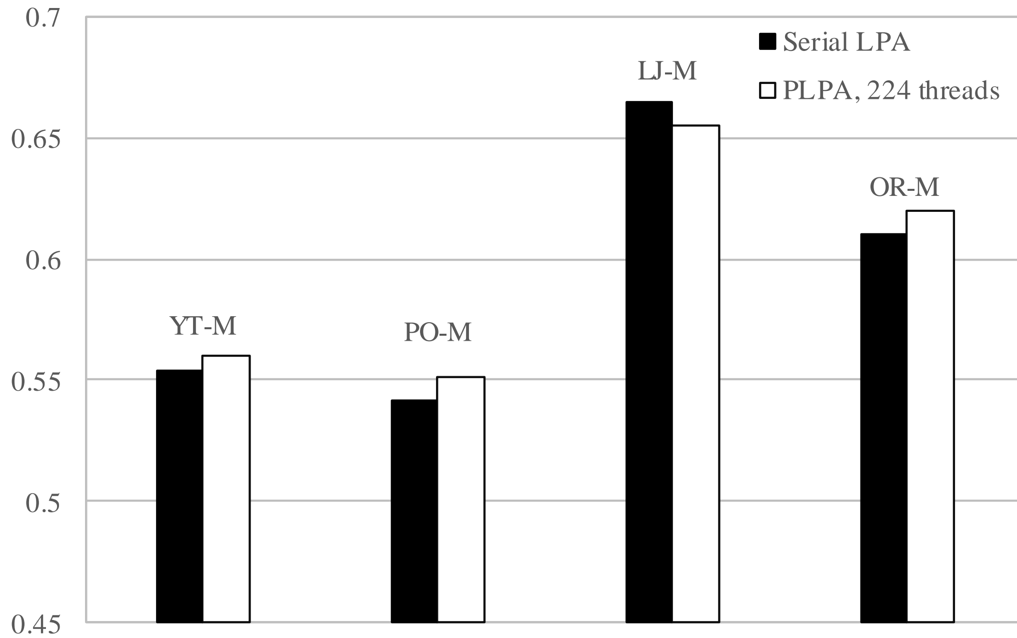

Figure 3.15: Modularity measurements for PLPA versus serial LPA.

To validate the accuracy aspect of the PLPA flow, we report modularity measurements in Figure 3.15. The modularity values deviate by a maximum of 2% when compared to the serial runs which is in line due to a non-deterministic nature of LPA. Therefore, our massively parallel LPA reproduced the qualities of the detected communities. We note that there is additional randomization on Phi with 224 threads through massive parallelization. Although, we aimed to minimize memory locking, it still occurs to some extent and this, along with the parallelization, leads to better clustering diversity as it was also noted in [32]. Contrary to higher complexity algorithms, the speed of LPA allows for multiple runs with a goal of finding the best clustering combination. This advantage is even greater with LPA running in parallel thereby significantly reducing a processing speed.

3.7 Label Propagation on Phi with massive networks

3.7.1 Motivation and a proposed algorithm

LPA on massive networks, e.g. (FR) in Table 3.2, is one of target applications for Phi. However, we were unable to perform community detection on Phi with FR due to the memory limitation. While the memory limitation will be mitigated in the next generation of Phi, there are several possibilities of processing FR on the current Phi with all of the choices focusing on graph partitioning. Ideally partitioning would split a problem into chunks that have little to no communication and, therefore, can be processed independently. However, this would be a community detection problem itself. Therefore, the partitioning have to be trivial. We adopt a divide and conquer approach and create equally sized slices of nodes. PLPA requires *L(V), D(V), NB(V)* and *A(V)* data structures, and they must support partitioning. We make an observation that only *L(V)* must be accessible by any node in order for PLPA to maintain a clustering quality. For example, FR has around 66 million nodes which means there shall be 264 MB available for *L(V).* In order to preserve processing speed we need another 264 MB to allocate for *A(V)*. We are left with two remaining structures that can be partitioned. Since *NB(V)* requires close to 7.2 GB and *D(V)* requires 264 MB, it is sufficient to split the dominating *NB(V)* in two equal slices to have FR processed in two steps. We profiled *NB(V)* and found that due to *D(V)* variations the node list must be split in a ¼ proportion in order to have balanced *NB(V)* partitions. Therefore, we note that partition points must be adjusted individually for each network based on *NB(V)* profiling, and this can be done at no cost during parsing. The slices are dispatched for Phi processing iteratively while keeping *L(V)* and *A(V)*

in Phi memory all the time. PLPA-M pseudocode is shown in Figure 3.16. Notably, this approach maintains the initial PLPA complexity of *O(M)* since the only change is the order of node (edge) processing.

Notably, it would be impossible to apply PLPA-M on graphs significantly larger than FR since the memory capacity of our card would be a non-resolvable bottleneck for other data structures in addition to *NB(V)*.

```
    Input:   G(V,E), Δupd > 0
    Output:  L(V)
1   Initialize: ∀v ∈ V: L(v) ← v, upd ← Δupd+1, A(v) ← 1
2   Phi allocation: ∀v ∈ G, L(v), A(v)
3   Partition D(v) and NB(v) ∈ partitions
4   while upd > Δupd do
5      upd ← 0
6      for ∀p ∈ partitions
7         Phi allocation: ∀v ∈ p, D(v), NB(v)
8         for ∀v ∈ p and D(v) > 0 and A(v)==1 do
9            L_temp = Max_L( ∑(u ∈ NB(v))
10           if L(v) ≠ L_temp then
11              L(v) ← L_temp
12              upd = upd + 1
13              ∀u ∈ NB(v), A(u) ← 1
14           Else
15              A(v) ← 0
16           end if
17        end for
18     end for
19  end while
20  send L(V) to Xeon for analysis
```

Figure 3.16: PLPA-M pseudocode. From [62].

3.7.2 Experimental results

We performed empirical evaluations of PLPA-M with FR with results shown in Figure 3.17. We utilized Phi with SMT turned on and scaled the processor from a single thread to a maximum 224 hardware threads on FR and achieved a maximum of 72 speedup. The speedup was limited by the overhead present due to extra offload time and synchronization. Overall run times shall be significantly improved on the next generation Phi product [39] as we have analyzed in section 3.5.2.

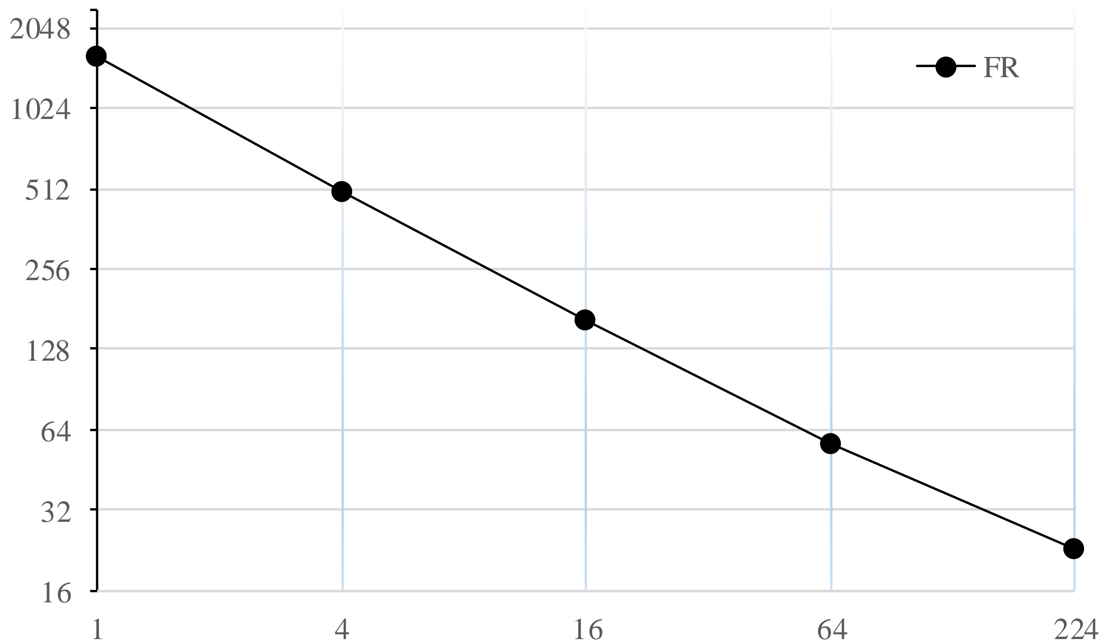

Figure 3.17: Strong scaling of PLPA-M (x-axis, thread count, base-4 log-scale) running time in seconds (y-axis, base-2 log-scale) on FR network. From [62].

3.8 Heterogeneous LPA and opportunities for massively parallel architectures

It is desirable to process networks concurrently both on Phi and Xeon thereby achieving a notable speedup. Such scheme could be also adopted to a distributed types of an architecture with multiple processing nodes. We present heurists for such flavor of LPA along with initial empirical work to assess scalability limits. The deficiencies present in the current generation of Phi noted in section 3.6.2 limit adoption of heterogeneous approaches. The main purpose of this section is to project algorithms for to the next generation of Phi that is launched in 2016 and, therefore, to set a base for future research..

The naïve divide-and-conquer strategy would be to split nodes proportionally to processing speeds of computing clusters so that all partitions can be processed asynchronously and within same time period. The issue with this method is that it resembles a synchronous LPA [17] that has several deficiencies reported in section 3.4.2 of this dissertation. Since the only condition for divide-and-conquer partitioning is to have equal slices, it is done randomly. The number of available partitions grows quadratically, and it is highly unlikely that a partition would happen in a preferred way which would minimize dependencies on labels in remaining partitions. On the contrary, it is very likely that partitioning would be slicing a graph in some average scenario which can the case illustrated in Figure 3.18. A random partitioning scheme would cause such deficiencies as lower clustering qualities and longer convergence times due to oscillation of labels. Hence, we do not consider this method to be an acceptable solution to adopt for hybrid and multi-node processing.

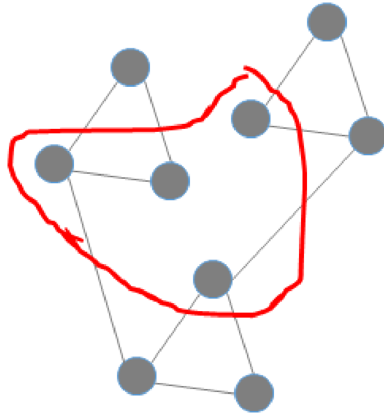

Figure 3.18: Partitioning that results in poor clustering and oscillation.

To explicitly address the deficiencies presented above, each node would have to have an access to the most recent node assignment. While there is no feasible solution to keep a shared data structure with efficient access from both Phi and Xeon processors [42], there is an existing way to efficiently transfer data between the two processors using their PCIe 2.0 interconnection. However, this would require a separate synchronization protocol supported both by hardware and software and this is beyond of the scope of this dissertation. Hence, we need to step back and refactor the PLPA.

The solution for the label coherency problem resides in proper partitioning of workloads so that they as much independent among themselves as possible. We assess there are two possibilities to achieve this goal and both of these options come with costs.

```
     Input:   G(V, E), Δupd > 0
     Output:  L(V)
1    Initialize: ∀v ∈ V: L(v) ← v, upd ← Δupd+1, A(v) ← 1
2    Coloring C(i) ∈ colors
3    Phi allocation: ∀v ∈ p, D(v), NB(v)
4    while upd > Δupd do
5       upd ← 0
6       for ∀c ∈ colors
7          process Xeon and Phi slices concurrently
8          for ∀v ∈ c and D(v) > 0 and A(v)==1 do
9             L_temp = Max_L( ∑(u ∈ NB(v))
10            if L(v) ≠ L_temp then
11               L(v) ← L_temp
12               upd = upd + 1
13               ∀u ∈ NB(v), A(u) ← 1
14            else
15               A(v) ← 0
16            end if
17         end for
18         synchronize L(V)
19      end for
20   end while
```

Figure 3.19: HLPA pseudocode.

The semi-synchronous approach [14] is considered in order to perform community detection simultaneously on Xeon and Phi. In the semi-synchronous propagation $\forall v, \forall u \in C_j$ there is no $e(v,u) \in G$. Therefore, the nodes that belong to the same color can be processed independently in a synchronous manner [17] since these nodes are not connected and, therefore, labels of their neighbors would not be updated at the same time. This feature eliminates the need for a shared memory access. In order to achieve a maximal speedup, each

color C_j would be split in a proportion that would guarantee equal time for all computing nodes to process their allocated workloads. While the overall complexity remains *O(N)* there are additional costs associated with graph coloring, synchronization steps after each color is processed, data transfers and label synchronization upon processing of each color. We provide a pseudo-code for this option in Figure 3.19.

Graph coloring has a *O(M)* complexity and can be efficiently parallelized [25, 66]. A proper coloring scheme could be adopted to provide sufficient work for both Xeon and Phi to justify hybrid processing. A greedy graph coloring scheme has been the most popular option. In this context, the goal is to have as fewer color as possible to minimize synchronization points. The bounds for the number of colors can assessed as $C_{min}=clique_{max}$ and $C_{max}=D_{max}$. A greedy algorithm such as the one presented in [25] optimally achieves $C_{max}=clique_{max}$. Table 3.5 illustrates empirical results of applying the greedy coloring scheme to our datasets and achieving numbers significantly less than the values of D_{max}.

The major cost associated with the semi-synchronous approach are synchronization points that will take place upon processing of each color. While distribution of color sizes does not follow power-law there is still a significant number of colors in the low range. This prevents ideal scalability of this approach due load balancing issues and underutilization of available threads as shown in Figure 3.20. In addition, another characteristic of the semi-synchronous approach is that LPA behavior is morphed into a semi-deterministic nature. This is since the only processing freedom that remains is the order in which colors are processed. This somewhat limits a quality of final clustering while producing more stable results when compared to the original LPA. Modularity and scalability results of running the datasets with

the greedy coloring scheme [25] are illustrated in Figure 3.21 and Table 3.5. Notably, HLPA

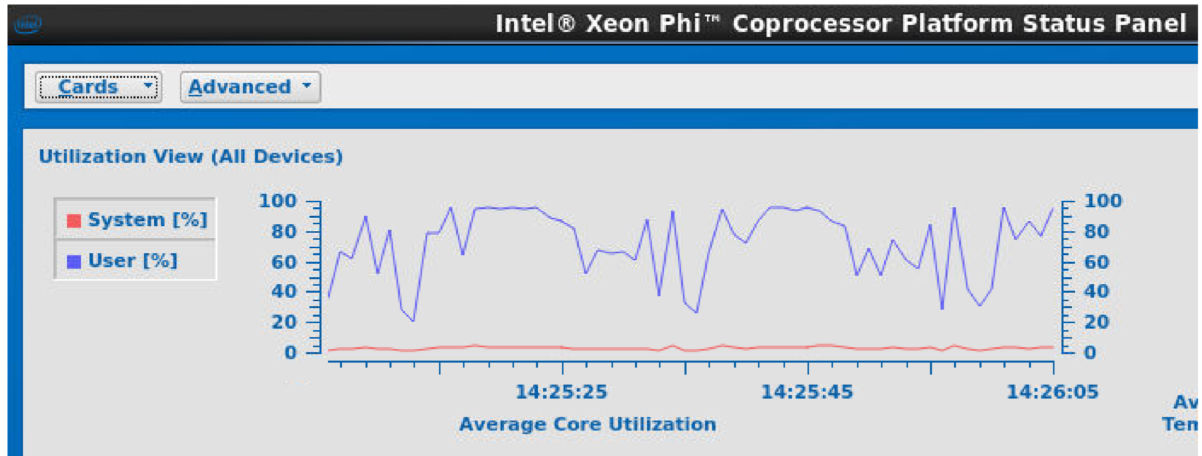

Figure 3.20: Suboptimal utilization of available threads due to synchronization points.

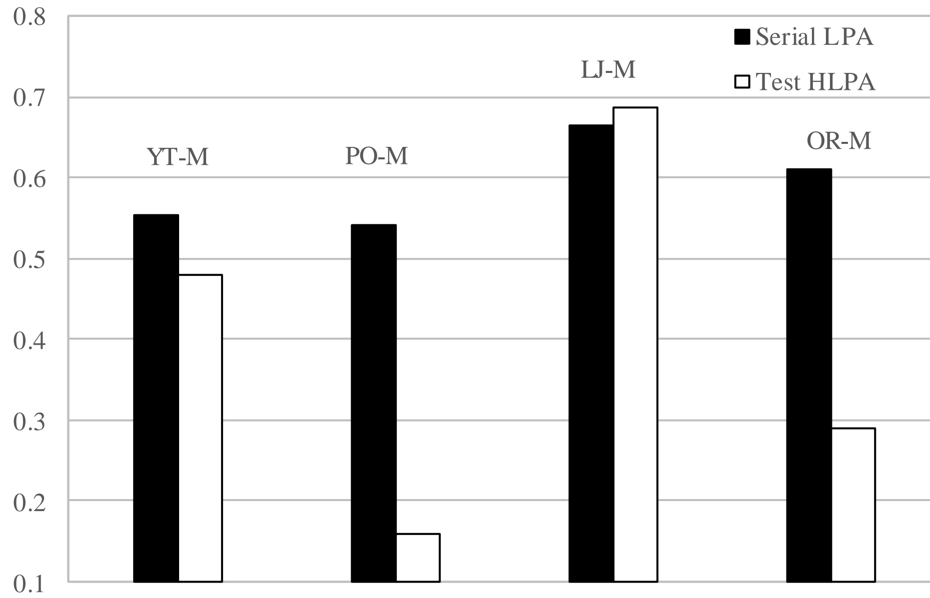

Figure 3.21: Modularity measurements for HLPA versus base LPA.

did not achieve the optimal scalability even on the largest network OR-M due to the issues previously discussed. FR network is excluded in these experiments due to the known memory limitation.

Table 3.5:
HLPA time and the first iteration scaling measurements (in seconds) on Xeon.

	YT-M	PO-M	LJ-M	OR-M
D_{max}	28754	14854	20334	33313
# of colors	39	41	325	127
Xeon 1 core	0.56	1.67	4.31	18.89
Xeon 4 cores	0.23	0.52	1.25	5.17
Speed up	2.43	3.21	3.45	3.65

The preferred option to efficiently utilize both Xeon and Phi processors. The current generation of Phi products has constraints noted in section 3.6.2 that prevent optimal execution of this method, hence we are projecting to the next Phi generation. It is possible to apply this approach to a higher number of computing nodes (a distributed memory architecture) by designing a coarse grain pre-processing algorithm that would behave similar to the well-known min-cut algorithm [65]. This approach would enable partitioning of a problem with a goal to minimize communication among partitions and, therefore, a dependency on labels in other partitions. The best known serial min-cut partitioning algorithms has a serial complexity close to $O(N \, LogN)$ [65]. Since the serial LPA complexity

is $O(M)$, it is necessary that the graph partitioning cost be better $O(M)$ in order for the hybrid (parallel) clustering algorithm such as HLPA to stay efficient. Various optimization strategies illustrated in this dissertation, such as parallelization, could be potentially applied to partitioning. With this, we set a base for future research that is in progress [67].

Chapter 4

Conclusions and future work

In the first half of this dissertation we proposed a simple and highly scalable plane sweep approach named Scan-List (SL). Despite this method having a higher order of sequential operation complexity than other well-known serial algorithms, its scalability allows it to surpass those algorithms when run in parallel. The proposed method was profiled against the best-known plane sweep method and was applied to tests generated using industrial EDA physical design tools. There is a need for future research and applications of this approach to a more general case of polygons. Having this algorithm properly tuned for the general case, would enable several high demand military and scientific application that deal with the surface scans.

In the second half of this dissertation, we introduced methods of utilizing Phi for community detection with LPA, a popular clustering method in network analysis. We tuned LPA to achieve linear scaling on a conventional Xeon architecture on large datasets. We illustrated PLPA, a modified LPA that was optimized for Phi. We also presented its modified version PLPA-M to run on massive networks with millions to billions of edges. We provided profiling and quality reports on running our algorithms on Phi with more than 200 threads on well-known publicly available networks. We achieved near linear speedups on Phi on large datasets while improving quality of the detected communities. We outlined the current Phi platform limitations that prevented PLPA from performing in an optimal way and provided

analysis how PLPA performance would be improved in the upcoming Phi based products. Finally, we provided heuristics for hybrid LPA flavors that can be adopted on a distributed type of architectures and Phi platforms with multiple cards. We illustrated initial empirical work of running LPA in semi-synchronous mode to allow for processing graphs on multiple computing nodes. Although, the current generation of Phi has limitations for the efficient heterogeneous execution noted in section 3.6.2, we can project this approach to the next more advanced Phi products to appear in 2016. These advanced possibilities of massively parallel network analysis with LPA is a base for future research [67].

Bibliography

[1] H. Lu, M. Halappanavar, A. Kalyanaraman, “Parallel heuristics for scalable community detection,” Parallel Computing, 2015, pp. 19-37, ISSN 0167-8191, DOI: 10.1016/j.parco.2015.03.003.

[2] J. Scott, “Social Network Analysis: A Handbook”, Sage, London, 2nd Ed, 2000.

[3] M. Newman, Proc. Natl. Acad. Sci. USA 98, 2001, 404–409.

[4] H. Jeong, B. Tombor, Albert, R., Oltvai, Z.N. and Barabasi, A.-L., Nature. London, 2000, 407, 651-654.

[5] D. Fell, A. Wagner, “The small world of metabolism,” Nature Biotechnology, vol. 18, no. 11, 2000, pp. 1121–1122.

[6] S. H. Strogatz, “Exploring complex networks,” Nature, London, 2001, 410, 268-276.

[7] S. Dorogovtsev, J. Mendes, “From Biological Nets to the Internet and WWW,” Oxford University Press, Oxford, 2003.

[8] M. Girvan, M. Newman, “Community structure in social and biological networks,” Proceedings of the National Academy of Sciences, 2002, vol. 99, no. 12, pp. 7821–7826.

[9] Facebook key facts, (accessed March 2015) [ONLINE]. Available: http://newsroom.fb.com/Key-Facts.

[10] S. Fortunato, “Community detection in graphs,” Physics Reports, 2010, 486(3-5):75–174.

[11] J. Fang, A. L. Varbanescu, H. J. Sips, L. Zhang, Y. Che, C. Xu, “An empirical study of Intel Xeon Phi,” CoRR, abs/1310.5842, 2013.

[12] OpenMP. (2008, May). “Openmp Application Program Interface” [Online]. Available: http://www.openmp.org/mp-documents/specs30.pdf.

[13] G. Cordasco, L. Gargano, “Community detection via semi-synchronous label propagation algorithms,” IEEE International Workshop on Business Applications of Social Network Analysis (BASNA), 2010, pp. 1–8. IEEE.

[14] X. Liu, T. Murata, “How does label propagation algorithm work in bipartite networks?” Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology, vol. 03, 2009, pp. 5–8. IEEE Computer Society.

[15] L. Subelj, M. Bajec, M., “Unfolding network communities by combining defensive and offensive label propagation,” Proceedings of the ECML PKDD Workshop on the Analysis of Complex Networks, ACNE 2010, pp. 87–104.

[16] R. Guimera, L. Amaral, Nature. London, 2005, 433, 895.

[17] U. Raghavan, R. Albert, S. Kumara, “Near linear time algorithm to detect community structures in large-scale networks,” Phys. Rev. E, September, 2007, Vol. 76, 036106, No. 3, pp.1–12

[18] J. Yang, J. Leskovec, “Defining and evaluating network communities based on ground-truth”, Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics, ACM, 2012, p. 3.

[19] M. Newman, “Fast algorithm for detecting community structure in networks,” Phys. Rev. E, Vol. 69, No. 6, 2004, 066133, pp.1–5.

[20] F. Comellas, A. Miralles, “A fast and efficient algorithm to identify clusters in networks,” Applied Mathematics and Computation, Vol. 217, No. 5, 2010, pp.2007–2014.

[21] R. Guimera, M. Sales-Pardo, L. Amaral, “Modularity from fluctuations in random graphs and complex networks,” Phys. Rev. E, Vol. 70, 2004, No. 2, pp.1–4.

[22] M. Newman, “Finding community structure in networks using the eigenvectors of matrices,” Phys. Rev. E, September, Vol. 74, 036104, No. 3, 2006, pp.1–22.

[23] M. Rosvall, C. Bergstrom, “An information-theoretic framework for resolving community structure in complex networks,” Proceedings of the National Academy of Sciences, USA, 2007, Vol. 104, pp.7327–7331.

[24] M. Newman, M. Girvan, “Finding and evaluating community structure in networks,” Phys. Rev. E, February, 2004, Vol. 69, 026113, No. 2, pp.1–16.

[25] E U. Catalyurek, J. Feo, A. Gebremedhin, M. Halappanavar, A. Pothen, “Graph coloring algorithms for multi-core and massively multithreaded architectures,” Parallel Computing 38, pp.576–594, 2012.

[26] V. Blondel, J. Guillaume, J. Lambiotte, E. Lefebvre, “Fast unfolding of communities in large networks,” Journal of Statistical Mechanics: Theory and Experiment, vol. 2008, no. 10, p. P10008.

[27] Q. Yang, S. Lonardi, “A parallel algorithm for clustering protein-protein interaction networks,” Proceedings of the 2005 IEEE Computational Systems Bioinformatics Conference Workshops (CSBW’05).

[28] D. Bader, K. Madduri, “Parallel algorithms for evaluating centrality indices in real-world networks,” Intl. Conf. on Parallel Processing, 2006, pages 539–550, Washington, DC.

[29] T. Hoeer, A. Lumsdaine, “A space efficient parallel algorithm for computing betweenness centrality in distributed memory,” 2010 International Conference High Performance Computing (HiPC), dec. 2010, pp. 1-10-20.

[30] A. Khlopotine, V. Jandhyala, D. Kirkpatrick, “A variant of parallel plane sweep algorithm for multicore systems,” Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 32, no. 6, 2013, pp. 966–970.

[31] U. Brandes, D. Delling, R. Gaerler, M. Hoefer, Z. Nikolovski, Z. Wagner, “On modularity clustering,” IEEE trans. Knowl. Data Eng. 20(2), 2008, 172-188.

[32] C. Staudt, H. Meyerhenke, “Engineering high-performance community detection heuristics for massive graphs,” Proceedings of the 2013 International Conference on Parallel Processing, Conference Publishing Services (CPS), 2013.

[33] X. Que, F. Checconi, F. Petrini, T. Wang, W. Yu, “Lightning-fast Community Detection in Social Media: A Scalable Implementation of the Louvain Algorithm,” Technical Report AU-CSSE-PASL/13-TRO1, 2013.

[34] E. Duriakova, N. Hurley, D. Ajwani, A. Sala, “Analysis of the Semi-synchronous Approach to Large-scale Parallel Community Finding,” Proceedings of the second ACM conference on online social networks, 2014, Pp.51-62.

[35] M. Ovelgönne, “Distributed community detection in web-scale networks,” University of Maryland, Tech. Rep, 2012.

[36] J. Soman, A. Narang, “Fast community detection algorithm with GPUs and Multicore architectures,” Proc. 25th IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2011, pp. 568–579.

[37] A. Duran, M. Klemm, “The Intel Many Integrated Core architecture,” Intel Corporation, 2011.

[38] Xeon Phi™ starter kit (accessed March 2015) [ONLINE]. Available: https://software.intel.com/en-us/xeon-phi-starter-kit.

[39] Knights Landing (accessed March 2015) [ONLINE]. Available: https://software.intel.com/sites/default/files/managed/e9/b5/Knights-Corner-is-your-path-to-Knights-Landing.pdf.

[40] Stanford Large Dataset collection (accessed March 2015) [ONLINE]. Available: http://snap.stanford.edu/data/index.htmlf.

[41] Intel® Parallel Studio XE (accessed March 2015) [ONLINE]. Available: https://software.intel.com/en-us/intel-parallel-studio-xe.

[42] Colfax International, “Parallel Programming and Optimization with Intel Xeon Phi Coprocessors,” ISBN: 978-0-9885234-1-8, 2013.

[43] M. I. Shamos and D. J. Hoey, “Geometric intersection problems,” Proc. 17th Annu. IEEE Symp. Foundations of Computer Science, pp.208-215, 1976.

[44] J. Bentley and T. Ottmann, “Algorithms for Reporting and Counting Geometric Intersections,” IEEE Trans. On Computers, C-28:643–647, 1979.

[45] H. Edelsbrunner, “Dynamic rectangle intersection searching,” Inst. for Informationsverarbeitung, TU Graz, Bericht 47 (February 1980).

[46] J. Bentley and D. Wood, “An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles,” IEEE Trans. On Computers, C-29:571-577, NO. 7, 1980.

[47] E. M. McGreight, “Efficient algorithms for enumerating intersecting intervals and rectangles," Rep. CSL-80-9, Xerox PARC, Palo Alto, California, (June 1980).

[48] D. T. Lee, “An Optimal Time and Minimal Space Algorithm for Rectangle Intersection Problems,” International Journal of Computer and Information Sciences, Vol. 15, No. 1, 1984.

[49] L.-T. Wang, Y.-W. Chang and K.-T. Cheng (Eds.), “Global and detailed routing” in Electronic Design Automation: Synthesis, Verification, and Test. Burlington, MA, USA: Elsevier / Morgan Kaufmann, 2009. ISBN: 0-1237-4364-8.

[50] H. W. Six and D. Wood, “The rectangle intersection problem revisited,” BIT, 20: 426-433, 1980.

[51] Intel Xeon Processor X7350, http://ark.intel.com/products/30796/Intel-Xeon-Processor-X7350-(8M-Cache-2_93-GHz-1066-MHz-FSB)

[52] Parallel Plane Sweep Algorithm for Multi-Core Systems, http://acelabuwee.org/resources/index.html

[53] GNU manual, Chapter 18. Parallel Mode, http://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html.

[54] Vicente H. F. Batista, David L. Millman, Sylvain Pion, and Johannes Singler, “Parallel geometric algorithms for multi-core computers,” Computational Geometry, vol. 43, pp. 663-677, 2010.

[55] Mark McKenney, Tynan McGuire, “A Parallel Plane Sweep Algorithm for Multi-Core Systems,” Proceedings of the ACM International Symposium on Advances in Geographic Information Systems, ACM SIGSPATIAL GIS 2009, pp. 392–395, 2009.UIT.

[56] E. S. Liu and G. K. Theodoropoulos, “An Approach for Parallel Interest Matching in Distributed Virtual Environments,” in Proceedings of the 2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, 2009, pp. 57- 65.

[57] A. L. Chow, “Parallel algorithms for geometric problems,” Ph.D. thesis, University of Illinois at Urbana-Champaign, 1980.

[58] A. Zomorodian, H. Edelsbrunner, “Fast software for box intersections,” Intl. J. Comp. Geometry Appl. 12 (1-2) (2002) 143{172.[16]

[59] Wang, Chang, and Cheng (Ed.), “Electronic Design Automation: Synthesis, Verification, and Test”, Morgan Kaufmann, 2009, chapter 12.

[60] Embarrassingly parallel problem, http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/AlgorithmStructure/EmbParallel.htm.

[61] D Ajwani, N. Sitchinava, N. Zeh, “Geometric Algorithms for Private-Cache Chip Multiprocessors,” ESA (2) 2010: 75-86.

[62] A. Khlopotine, A. Sathanur, V. Jandhyala, “Optimized Parallel Label Propagation based Community Detection on the Intel® Xeon Phi™ Architecture,” 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2015).

[63] T. N. Mudge, R. A. Ratenbar, R. M Lougheed, and D. E. Atkins, "Cellular lmage Processing Techniques for VLSI Circuit Layout Validation and Routing," ACM IEEE Nineteenth Design Automa-Conference Proceedings, pp 537-543.

[64] W. H. Kao, C.-Y. Lo, M. Basel, and R. Singh, "Parasitic extraction: Current state of the art and future trends," Proc. IEEE, vol. 89, no. 5, pp. 729–739, May 2001.

[65] G. Italiano, Y. Nussbaum, P. Sankowski, C. Wullf-Nilsen, "Improved algorithm for min cut and max flow in undirected planar graphs," Proceedings of the forty-third annual ACM symposium on Theory of computing Pages 313-322.

[66] G. Rokos, Gerard Gorman, P. Kelly, "A fast and scalable graph coloring algorithm for multi-core and many-core architectures", May, 2015.

[67] A. Khlopotine *et. al*, "Heterogeneous Label Propagation Algorithm on the Intel® Xeon Phi™ Architecture," to be submitted