Opportunities for Flow Based Parallelism —— A Case Study of Three Open Source Applications

Donghun Lee*
Department of Computer Science
University of Illinois at Urbana Champaign

Abstract

This paper investigates open source applications to seek for opportunities of flow-based parallelization. Domain of streaming application are targeted such as sentence parsing, image processing and data packet investigation. Component flow of application is identified and profiling is conducted to discover bottleneck of program executions. After parallelization attempt, challenges of flow-based parallelization are identified and discussed.

Keywords: flow-based parallelism, bottleneck

1 Introduction

Recent advance of technology enabled us to utilize mult-core processors and highlighted the need of effective concurrent programming manner. Flow based applications have their processes in a sequential order. Each black box stages are separated and they exchange data after their individual process. Because of the fact that each stage depends on data from previous stage, order of stages are important. When a bottleneck of program execution is found to be a particular stage or sequence of stages, parallelizing that particular stage or sequence of stages improves overall performance of the program. Bottleneck is the component which takes most of execution time.

Open source applications open source code to public and allow users to study, customize, improve and also distribute the software. Open source applications are adapted in this paper to have an access to source code of creditable applications. Three streaming applications are introduced in this paper. They are syntactic sentence parser JLinkGrammar, content based image retrieval library LIRE and network intrusion detecting application Snort.

This work is motivated to catalog study of parallel patterns from version control of open source applications and manual parallelization. However paper concentrates on manual parallelizations and opportunities of parallelizations since three open source applications didn't show any parallel pattern.

Section 2 talks about similar methodology which is applied to each open source application. JLinkGrammer is discussed in section 3, LIRE in section 4 and Snort in section 5.

2 Methodology

The goal of this work is to explore opportunities of flow-based parallelism along flow-based applications. Three widely used flow-based open source applications are investigated which are JLinkGrammar, LIRE and Snort. JLinkGrammar and LIRE are written in Java and Snort is written in C. This paper studies system architecture and component flow of each application and finds bottleneck of application execution. We are concerned with bottleneck of the applications because we can achieve performance improvement by parallelizing bottleneck. YourKit Java Profiler 10.0.6 is used to profile Java written applications and point out bottleneck. When bottleneck of application is found, source codes are investigated to see parallelization opportunities. If they are able to parallelized, it is said that the application follows flow-based parallelism. If they are not be able to parallelized, obstacle of parallelization is identified.

*e-mail: lee329@illinois.edu, Undergrad

```
linkparser>
Last week I saw a great movie

Found 1 linkage (1 had no P.P. violations)
  Unique linkage, cost vector = (UNUSED=0 DIS=1 AND=0 LEN=12)

                      +-------Os-------+
                      |   +-----Ds-----+
   +--DTi-+CO*n+Sp*i+  |   +---A---+    |
   |      |    |    |  |   |       |    |
last.a week.t I.p saw.v a great.a movie.n
```

Figure 1: *JLinkGrammar execution for a valid sentence.*

3 JLinkGrammar

3.1 Description

JLinkGrammar is a synthetic English parser which examines correctness of English grammars. JLinkGrammar is a Java version of C written program Link Grammar Parser which is originally developed by Carnegie Mellon University. When a sentence is given to the program, it decides whether the sentence has a valid *linkage* or not. Then the program outputs the linkage that it found, showing the words that are *linked* together and the type of link between them. [Temperley] Syntactic structure is presented as a linkage which are collections of valid links. Links are different types of connection between English connectors. Over hundreds of links are locally defined to classify and and construct sentences in mathematical structure.

Figure 1 shows an example of program execution. When a valid sentence "Last week I saw a great movie" is entered, a syntactic structure is constructed saying *Found 1 linkage*. Each pair of word is connected with link and completely valid links forms linkage which are graphically shown in the example. We can observe that link A connects pre-noun adjective *great* to noun *movie*. D connects determiners to noun. D connector is extended to Ds meaning singularity. O connects transitive verbs to direct or indirect objects. [Sleator] O connector also keeps information of singularity by extension to Os. Likewise, every part of sentence is dissected and classified based on their link definition.

When an invalid sentence is parsed, JLinkGrammar detects which part of the sentence is grammatically wrong. Previous example sentence is parsed with slight modification. Noun *week* is modified to *dog* in Figure 2. The application outputs *No complete linkages found* when it decides invalid sentence is parsed and it highlights grammatically incorrect part *dog* with *square brackets*.

```
linkparser>
Last dog I saw a great movie
No complete linkages found.

Found 1 linkage (1 had no P.P. violations) at null count 1
  Unique linkage, cost vector = (UNUSED=1 DIS=0 AND=0 LEN=13)

                    +-------Os-------++
                    |   +-----Ds-----++
   +----CO----+Sp*i+   |    +---A---++
   |          |    |   |    |       |
last.a [dog] I.p saw.v a great.a movie.n
```

Figure 2: *JLinkGrammar execution for an invalid sentence.*

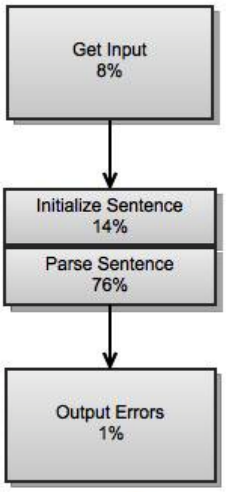

Figure 3: *Flow graph of JLinkGrammar execution.*

3.2 Bottleneck

3.2.1 Methodology for finding bottleneck

JLinkGrammar provides batch mode input. Batch mode enables user to input a file of sentences and run them through the program all at once. Original developer provides 900 tested sample input sentences. Each sentence is parsed to the application and accurately judged as correct or incorrect sentence. Since the execution time taken to parse 900 sentences is less than five seconds, input sentences are duplicated to gain enough time to profile the application while execution. So sample sentences are copied 50 times to make 45000 inputs and they takes about 90 seconds to completely judge which sentences are valid and which are not. Sample is run ten times to gain accurate output.

3.2.2 Flow and bottleneck

Figure 3 shows flow graph of program execution. Most of running time is consumed when sentences are parsed. During the stage where a sentence is parsed two steps are done before parsing. They are *expression_prune* and *prepare_to_parse*. They takes 13% and 47% of total execution time respectively. When numbers of sentences are input to the program at once, single sentence undergoes parsing process and rest of the sentences are waiting for their turns.

3.2.3 Expression_prune

JLinkGrammar uses disjunctive form to analyze grammars. In disjunctive form, each word of the grammar has a set of disjuncts associated with it. Each disjunct corresponds to one particular way of satisfying the requirements of a word. During stage of expression pruning, program deletes irrelevant disjuncts which contain a connector that does not match any other connector on a word in the sequence. Therefore, irrelevant disjuncts can be deleted without affecting the set of linkages. [Sleator]

3.2.4 Prepare_to_parse

All the necessary pruning and building of a structure is done in this stage. Disjuncts are built out of expressions and duplicates are deleted. Many other forms of pruning are processed such as gentle power pruning, power pruning, conjunction pruning and fat disjunct pruning.

3.3 Parallelization opportunity

Embarrassingly parallel parallelization is practiced but it did not work. Embarrassingly parallel workload can separate into parallel tasks where there exists no dependency (or communication) between each other. Therefore little or no effort is required to separate the problem into a number of parallel tasks.

When program runs in batch mode, there is a big *while loop* with a condition of reading a sentence from a file. When the program is able to read a sentence from a file, it executes the codes in the while loop. I extracted codes in the while loop to a method *doWhile* and created four threads running doWhile method. When precessing each sentence is an independent procedure, executing in four parallel tasks will improve the performance of program execution by four times. If the program run without interruption and correct output to valid and invalid sentences, it means the program is embarrassingly parallel.

However execution of parallelized program gives unstable output. Their output is incorrect and sentences are parsed incorrectly. Sometime *java.lang.ArrayIndexOutOfBoundsException* occurs from certain thread and sometime program stops and fall into deadlock state or race condition.

JLinkGrammar is not embarrassingly parallel application. Even though each sentence is independent, sentence.java class contains 29 static variables and 23 static methods. It was hard to eliminate dependencies between each sentence to treat them in same execution. Design of JLinkGrammar is not concurrency-programming-friendly.

Listing 1: *Parsing sentences from a file in batch modet*

```
public static void doIt(String arg[]) throws IOException {
  InitializeVars(arg);
  while (GlobalBean.fget_input_string(input_string,
    opts.input, opts.out, opts)) {

        . . .

        sent = new Sentence(input_string.toString(),
        dict, opts);

        . . .

        num_linkages = sent.sentence_parse(opts);
  }
}
```

3.4 Discussion

In Thies's paper [Thies et al. 2007] 197.parser is parallelized under restricted conditions and restricted inputs. 197.parser is an old version of Link Grammar Parser which is original program of

JLinkGrammar. During parallelization process of 197.parser, they made three adjustments. The program contained loop-carried dependences due to the program's implicit use of uninitialized memory locations. The program allocated space for a struct and later copies the struct (by value) before all of the elements have been initialized. They eliminated these dependence reports by initializing all elements to a dummy value at the time of allocation. [Thies et al. 2007]

Parallelization of 197.parser shows opportunity of parallelization of Link Grammar Parser which is recent version of 197.parser. This indicates opportunity of parallelization of JLinkGrammar which is Java version of C written program Link Grammar Parser. Therefore comparison of 197.parser, Link Grammar Parser and JLinkGrammar is conducted.

We observed that structure of code has improved from 197.parser to Link Grammar Parser v4.1b. Also we observed that conversion from Link Grammar Parser v4.1b to JLinkGrammar contains structural modifications. Five basic 'types' in the link parser API is defined in api-structures.h file. These structure are Dictionary, Parse_Options, Sentence, Linkage, PostProcessor and they are all required to analyze single sentence. Different from java version using numbers of static variables, C version is not using single static variable for these basic 'types'.

C version had helper function 'put_opts_in_local_vars' in command_line.c file to store global options in local. Since the function is call before while loop of main and we know only while loop is parallelized, the fact that this function doesn't exist in java version is insignificant.

4 LIRE

4.1 Description

LIRE (Lucene Image REtrieval) is a Java library for Content Based Image Retrieval (CBIR). [Lux and Chatzichristos 2008] *Content based* is an opposed technique of *concept based*. Content based indicates that search system analyzes a target image based on contents of the image such as colors, shapes, textures or any other features that can be retrieved from the image. Opposed to content based, concept based search system analyzes a target image based on metadata of the image such as tags, dates, location, etc.

LIRE extracts image features from raster images and stores them in a Lucene index for later retrieval. [Lux and Chatzichristos 2008] There exists 11 different image features that can be applied to index a set of images. They are:

- JpegCoefficientHistogram
- Three Tamura features
- AutoColorCorrelation
- FCTH
- JCD
- Gabor
- MPEG-7 ColorLayout
- MPEG-7 EdgeHistogram
- Simple RGB color histograms
- MPEG-7 ColorLayout

These feature information is used to index and retrieve images. All available features or combinations of popular features can be used.

```
1. document1 = {path=photo1.jpg, dominantcolor=120,12,34}
2. document2 = {path=photo2.jpg, dominantcolor=128,244,95}
3. document3 = {path=photo3.jpg, dominantcolor=1,39,232}
```

Figure 4: *Lucene index of example images*

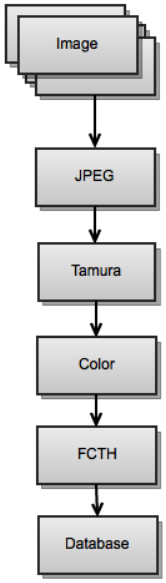

Figure 5: *Indexing procedure of LIRE*

In general Lire just takes numeric images descriptors, which are mainly vectors or sets of vectors, and stores them inside a Lucene index as text along with the image path within a Lucene document. So its like a very primitive database. [int] Here is a simplified example. Assume that LIRE extracts dominant color of RGB values from three images and stores them in Lucene. The index for three images is shown in figure 4. Each image is stored in *document*. We can observe that image path is specified and their figures are stored. When LIRE searches index, it opens every single document within the index, parses the vector and compares it to the query vector (e.g. with an L1 distance). The best matching documents are stored in a result vector along with the distance. [int]

4.2 Bottleneck

4.2.1 Flow and bottleneck

In a CBIR system, the main bottlenecks to performance are the indexing and retrieval stages. [Chen] This paper concentrates on indexing stage of LIRE. LIRE provided example code for indexing stage in /LIRE/src/main/java/net/semanticmetadata/lire/example/LireIndexingExample.java. When a set of 100 images is specified, LIRE processes indexing in flow-based manner shown in figure5. For each image in a image set, first LIRE extracts JpegCoefficientHistogram (JPEG) feature. Then extracted JPEG feature is used to extract three Tamura features. Then extracted Tamura features are used to extract AutoColorCorrelation (Color) feature. Then extracted Color feature is used to extract FCTH feature. Details of features are not important here.

The bottleneck in the indexing stage comes from the number of features extracted and the number of images that need to be analyzed. [Chen] Features used to extract from image are not independent. In order to extract a particular feature, information from

previous feature is needed. So the time taken to extract desired feature from an image depends on numbers of features extracted. Also because of the fact that indexing is linearly processed, total execution time would depend on numbers of images to be analyzed.

4.3 Parallelization opportunity

Pipeline parallelism is not encouraged here. When separate thread for each feature extraction stage is created, buffer stage should be introduced between each stage to control dependency between stages. Unfortunately, such an approach requires the developer to deal with the low-level threading and concurrency constructs resulting in copious boilerplate code that might hide concurrency problems. Moreover, the developer is burdened with the task of load balancing the different stages to achieve good performance. [Chen]

Instead of pipeline parallelism, we can apply flow-based parallelism as shown in figure 6. After JPEG feature is extracted from image 1, Tamura feature is extracted using JPEG feature which is extracted from previous step. Meanwhile, JPEG feature of image 2 can be extracted and so on. Likewise, each stage doesn't have to process a single feature at a time.

In order to deal with dependency, *LinkedBlockingQueue* is used which inherits thread-safe data structure *BlockingQueue*. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control. [BQ] Also elements inside blockingQueue are accessible from any threads. Keyword *put* is used to add elements and *take* is used to remove and delete from queue in this case. *CountDownLatch* and *poison pill* is used to terminate threads. CountDownLatch is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. [BQ]

Original code is shown in listing 2. We can notice that *docJPEG* is needed to obtain *docTamura* and so on. Listing 3 shows preparation for flow-based parallelism. Poison pill is set to string *This is end* and CountDownLatch is set to 6. Numbers of SynchronousQueue is defined too. *BufferedImage* and *imagePath* are defined for each stage because they are different from each stage when flow-based parallelism is applied. Likewise, SynchronousQueue for *docJPEG* is defined for *docTamura* and so on.

Listing 2: *Original code of indexing stage*

```
for (String imagePath : imagePaths) {
  BufferedImage bufferedImage = ImageIO.read(
                new FileInputStream(imagePath));

  Document docJPEG = JPEGCoefficientHistogramExtractor()
                .createDocument(bufferedImage, imagePath);
  Document docTamura = tamuraExtractor()
                .createDocument(docJPEG,
                                bufferedImage, imagePath);
  Document docColor = autoColorCorrelogramExtractor()
                .createDocument(docTamura,
                                bufferedImage, imagePath);
  Document docFCTH = FCTHExtractor()
                .createDocument(docColor,
                                bufferedImage, imagePath);
  indexWriter.addDocument(docFCTH);
}
```

Listing 3: *Initialization of SynchronousQueue*

```
private static final String POISON_PILL
        = "This is the end";
CountDownLatch latch = new CountDownLatch(6);
BlockingQueue<Document> docJPEG
        = new LinkedBlockingQueue<Document>();
BlockingQueue<Document> docTamura
        = new LinkedBlockingQueue<Document>();
BlockingQueue<Document> docColor
        = new LinkedBlockingQueue<Document>();
BlockingQueue<Document> docFCTH
        = new LinkedBlockingQueue<Document>();
```

Listing 4 shows pseudo implementation of flow-based parallelism. Information of JPEG is needed in order to extract Tamura feature. Also it is defined above that bufferedImage and imagePath are set to each stage independently. So JPEG document, bufferedImage and imagePath is *taken* and generated Tamura document is *put* in synchronousQueue named docTamura which is going to used in next stage, Color. Likewise runnable for each stage is set and they are run in parallel.

Listing 4: *Pseudo-code of thread for each stages of* docTamura

```
Runnable GenDocTamura = new Runnable(){
  @Override
  public void run() {
  while(true){
    try{
    Document JPEG = docJPEG.take();

    String imagePath = Tamura_Imagepath.take();
    if (imagePath == POISON_PILL) {
        latch.countDown();
        break;
    }

    BufferedImage img = Tamura_bufferedImg.take();
    Document doc = tamuraExtractor().createDocument(
                        JPEG,img, imagePath);
    docTamura.put(doc);

    } catch (Exception e) {
      e.printStackTrace();
    }
    }
  }
};
```

4.4 Experimental results

Version	Average of 100 image process (ms)
Original LIRE	32119
Parallelized LIRE	21745

Figure 7: *Mean processing time of 100 images 10 times on original and parallelized version of LIRE*

I used original version of LIRE and parallelized version of LIRE to process 100 images 11 times each and discarded their first attempts. Experiment is executed on 2.3GHz Intel Core i5 and a 1.6.0_29 version of JVM with 1Gb of ram. Figure 7 shows there there is about 1.5x speed up of execution time of flow-based parallel version over original version.

4.5 Discussion

Experiment shows that LIRE is well structured and it is easily parallelized in flow-based manner. Parallelized version showed about 1.5x of performance improvement over original serial version.

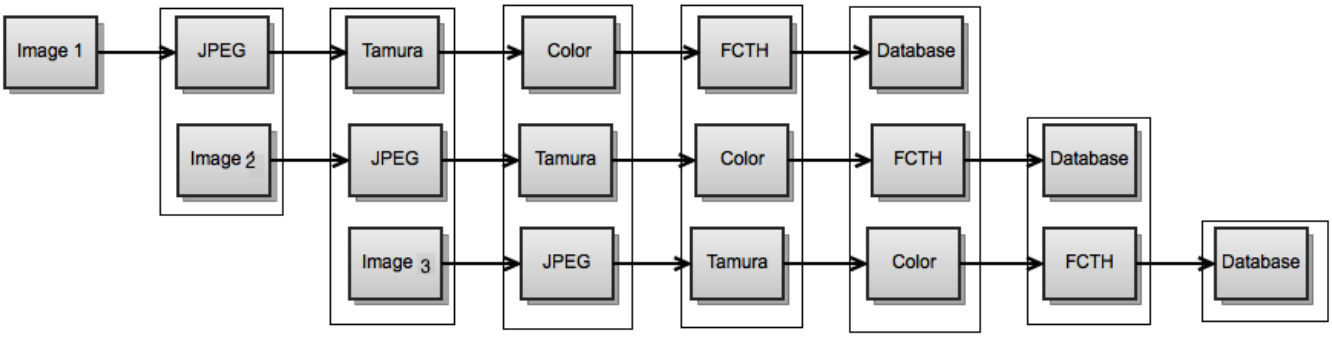

Figure 6: *Mechanism of flow-based parallelism for LIRE*

5 Snort

5.1 Description

Snort is a open source software developed by Sourcefire to detect and prevent network intrusions. Intrusion detection/prevention system (IDS) dynamically monitors incoming network traffic to discover unauthorized or suspicious accesses. Snort can perform protocol analysis and content searching/matching. Snort can also detect variety of attacks and probes, such as buffer overflows, stealth port scans, Common Gateway Interface (CGI) attacks, Server Message Block (SMB) probes, operating system fingerprinting attempts, and much more. [et al 2008]

Snort can be configured in three main modes: sniffer, packet logger, and network intrusion detection. Sniffer mode simply reads the packets off the network and displays them in a continuous stream on the console. Packet logger mode logs the packets to the disk. Network intrusion detection mode is the most complex and configurable, allowing Snort to analyze network traffic for matches against a user-defined ruleset and to perform one of several actions, based on what it sees. [et al 2008] This paper concerns about network intrusion detection mode because most complex component has most opportunity of parallelization.

Snort can be divided into five major components that are each critical to intrusion detection shown in figure8. The first is the packet capturing mechanism. Snort relies on an external packet capturing library (libpcap) to sniff packets. After packets have been captured in a raw form, they are passed into the packet decoder. The decoder is the first step into Snort's own architecture. The packet decoder translates specific protocol elements into an internal data structure. After the initial preparatory packet capture and decode is completed, traffic is handled by the preprocessors. Any number of pluggable preprocessors either examine or manipulate packets before handing them to the next component: the detection engine. The detection engine performs simple tests on a single aspect of each packet to detect intrusions. The last component is the output plugins, which generate alerts to present suspicious activity. [Koziol 2003]

5.2 Bottleneck

Snort is a huge program so bottleneck is not identified under limited time.

5.3 Parallelization opportunity

According to the paper by Intel [Intel], Snort can be successfully parallelized with pipelining and flow-pinning manner. Main strategy is to increase *locality of reference* to utilize caches from multi-core processor. Locality of reference presents in a request stream whereby bursts of references are made in the near future to objects referenced in the recent past. Increased locality of reference should yield performance improvements for demand-driven caching that exploits recency of reference. [Vanichpun and Makowski 2004]

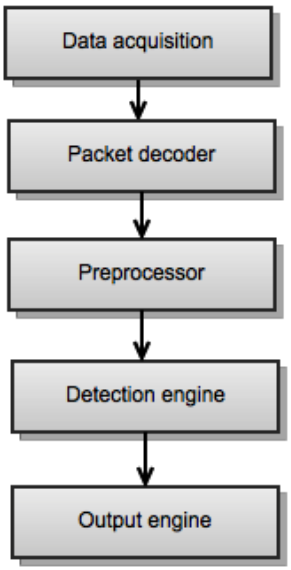

Figure 8: *Flow graph of Snort.*

Data acquisition stage is executed on one core. After data capturing, packets are distributed with flow-pinning manner and each data packets are processed concurrently. Flow-pinning manner is grouping and concentrating individual TCP flows to single core to improve locality of reference during packet processing. Applying pipelining and flow-pinning techniques with four execution cores results more than 6.2 speed up compare to single core system.

6 Discussion

This work is motivated from making *catalog* of Interactive source-to-source transformation by Nicholas Chen [Chen]. Transformations work is developing tool to help developers parallelizing their works. Catalog which is basis of transformation work is systematically catalog the parallel patterns and manual transformations that developers perform by studying a range of open source flow-based applications and their evolution. [Chen]

7 Conclusion and future work

This work shows flow-based application follows flow-based parallelism. There is well structured application like LIRE which is easily parallelize to benefit performance improvement. Parallelized version of LIRE showed about 1.5x performance improvement over original serial version of LIRE. JLinkGrammar is not well structured in terms of concurrency-programming-friendly. JLinkGrammar uses too many global variable and method so it is hard to parallelize. This contributes *catalog* of *Transformation* work. [Chen]

Opportunity of parallelization for Snort is not clearly discovered. Deeper analysis of Snort's internal implementation can be done to find out bottleneck of Snort execution. Also JLinkGrammar can be restructured to apply flow-based parallelism and study performance improvement.

8 Acknowledgements

I would like to thank Nicholas Chun Y Chen for his mentoring, clear explanations and guidance of research directions. Thanks to Prof. Lawrence Angrave and Ashish Vulimiri for their advice on project.

References

Java Platform, Standard Edition 6 API Specification. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html.

CHEN, N. *SOURCE-TO-SOURCE TRANSFORMATIONS FOR OBJECT-ORIENTED FLOW-BASED APPLICATIONS*. PhD thesis, University of Illinois at Urbana-Champaign.

ET AL, M. K. 2008. *How to Cheat at Securing Linux*. MA: Syngress Publishing Inc..

lire:lireinternals. http://www.semanticmetadata.net/wiki/doku.php?id=lire:lireinternals.

INTEL. Supra-linear Packet Processing Performance with Intel Multi-core Processors.

KOZIOL, J. 2003. *Intrusion Detection with Snort*. Sams; 2nd edition.

LEE, E. A. 2006. The Problem with Threads. *Computer 39* (May), 33–42.

Lirewiki. http://www.semanticmetadata.net/wiki/.

LUX, M., AND CHATZICHRISTOS, S. A. 2008. Lire: lucene image retrieval: an extensible java cbir library. In *MM '08 Proceedings of the 16th ACM international conference on Multimedia*.

RUL, S., VANDIERENDONCK, H., AND DE BOSSCHERE, K. 2006. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Computing 36* (September), 531–551.

SLEATOR, D. Index to Link Grammar Documentation. http://www.link.cs.cmu.edu/link/dict/index.html.

Java Platform, Standard Edition 6 API Specification. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/SynchronousQueue.html.

TEMPERLEY, D. An Introduction to the Link Grammar Parser. http://www.link.cs.cmu.edu/link/dict/introduction.html.

THIES, W., CHANDRASEKHAR, V., AND AMARASINGHE, S. 2007. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, 356 –369.

VANICHPUN, S., AND MAKOWSKI, A. M., 2004. Comparing strength of locality of reference - popularity, majorization, and some folk theorems.